Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Desarrollo de una herramienta de visualización e interacción virtual de usuarios captados por sensores Kinect v2

Autor: Alba Fortes Martínez

Tutor: Javier Macías Guarasa

2018

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Desarrollo de una herramienta de visualización e interacción virtual de usuarios captados por sensores Kinect v2

Autora: Alba Fortes Martínez

Tutor: Javier Macías Guarasa

Tribunal:

Presidente: Marta Marrón Romera

Vocal 1º: Juan Carlos García García

Vocal 2º: Javier Macías Guarasa

Calificación: .....

Fecha: .....

A mis padres, por su paciencia e incondicional apoyo.

"No todo lo que cuenta puede ser cuantificado, y no todo lo que puede ser cuantificado cuenta."
Albert Einstein

Resumen

El objetivo de este trabajo es la implementación de una biblioteca que permita el reconocimiento de las personas presentes en un espacio físico y el rastreo de sus movimientos corporales, así como su interacción con objetos presentes en un entorno virtual. Para ello, se ha utilizado una cámara de tiempo de vuelo (ToF), Kinect v2, y las bibliotecas OpenNI v2 y NiTE v2. La librería desarrollada identifica las articulaciones de las personas presentes mediante los datos de profundidad proporcionados por la Kinect y las bibliotecas previamente mencionadas para, posteriormente, mostrarlas en un entorno virtual.

Palabras clave: Entorno virtual, visualización de usuarios, detección de esqueleto corporal, cámaras de tiempo de vuelo.

Abstract

The aim of this work is the implementation of a library that permits the identification of the people present in a real-world space and the tracking of their corporal movements, as well as their interaction with objects present in a virtual environment. For this, it has been used a time-of-flight (ToF) camera, Kinect v2, and the middlewares OpenNI v2 and NiTE v2. The developed library identifies the joints of the people present throughout the depth-stream data provided by the Kinect and the previously mentioned middleware as to, subsequently, display them in a virtual environment.

Keywords: Virtual environments, user visualization, skeleton tracking, time-of-flight cameras.

Resumen extendido

Este trabajo de fin de grado (TFG) tiene como objetivo la implementación de una biblioteca, que permita la detección de las personas presentes en un determinado espacio físico y su visualización sobre un entorno virtual, mediante el empleo de una cámara de tiempo de vuelo ó Time of Flight (ToF), en concreto la Kinect v2.

Este proyecto completa el trabajo desarrollado en [1], en el cual se utilizo una Kinect v1. Así como implementa parte del trabajo realizado en [2], específicamente en este trabajo se hace uso del entorno virtual desarrollado inicialmente para el proyecto citado.

Para el cumplimiento de este objetivo se parte de los datos proporcionados por la Kinect v2, una cámara ToF. Desarrollando los algoritmos necesarios para la interpretación de los datos proporcionados en el lenguaje de programación C/C++. Esta nueva librería que se va a implementar, a su vez, se incorpora a un proyecto previo ([2]), que hace uso de la interfaz Open Graphics Library (OpenGL) para generar el entorno virtual sobre el que se muestran los datos obtenidos.

El sistema desarrollado tiene como objetivo principal la identificación y seguimiento de los esqueletos de cada usuario presente y su visualización dentro del entorno virtual desarrollado. Para ello se emplea la cámara Kinect v2, que proporciona información de profundidad, de infrarrojos y de color, de las cuales se muestra un ejemplo en la figura 2. En este proyecto solo se emplea la información de profundidad como medida para la preservación de la privacidad de la identidad de las personas.

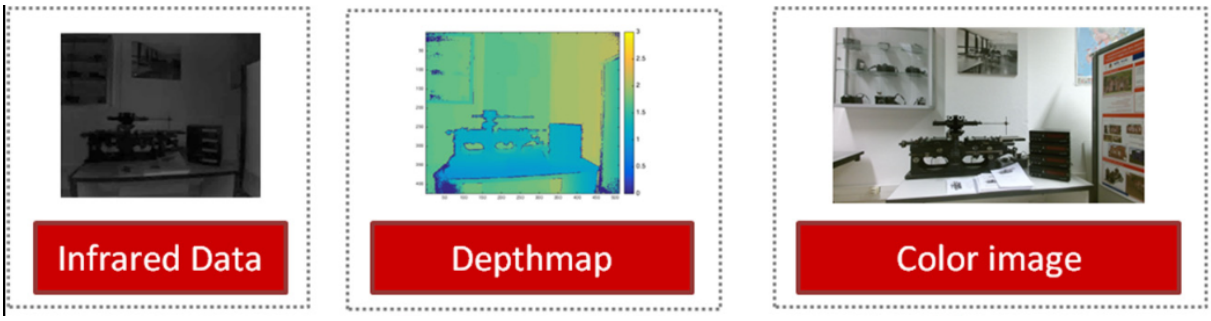


Figura 2: Ejemplo (extraído de [3]) de los tres tipos de imágenes obtenibles mediante el uso de una Kinect v2. A la izquierda la imagen de infrarrojos, en el centro la imagen de profundidad y a la derecha la imagen de color.

Esta cámara emplea la tecnología ToF, esto implica que obtiene la información de profundidad mediante la obtención de la diferencia de fases entre el haz de luz emitido y el recibido, cuya comparación permite obtener el tiempo que le ha tomado a este haz de luz modulada en regresar hasta el sensor.

Para la realización del proyecto también se han hecho uso de los middlewares Open Natural Interaction (OpenNI) y Natural Interaction Technology for End-user (NiTE), ambos en sus segundas versiones [4]. OpenNI [5] proporciona una interfaz de programación de aplicaciones, ó Application Programming

Interface (API), para aplicaciones que requieran el uso de interacción natural para la interacción con el usuario. De esta forma, OpenNI provee una interfaz de comunicación tanto para los sensores hardware compatibles, ya sean de audio, video o profundidad, como para middlewares de percepción, como NiTE, permitiendo de esta forma una interacción fluida entre ellos. NiTE [6], es un middleware desarrollado por PrimeSense, que actúa como motor de percepción del mundo que lo rodea y de la interacción del usuario con este. NiTE esta compuesto de múltiples funciones que le permiten alcanzar diferentes objetivos, entre ellos, el rastreo del esqueleto corporal, la identificación de gestos ó poses concretas y la distinción de los usuarios presentes del entorno.
En la figura 3 se puede observar la interrelación entre los distintos componentes mencionados.

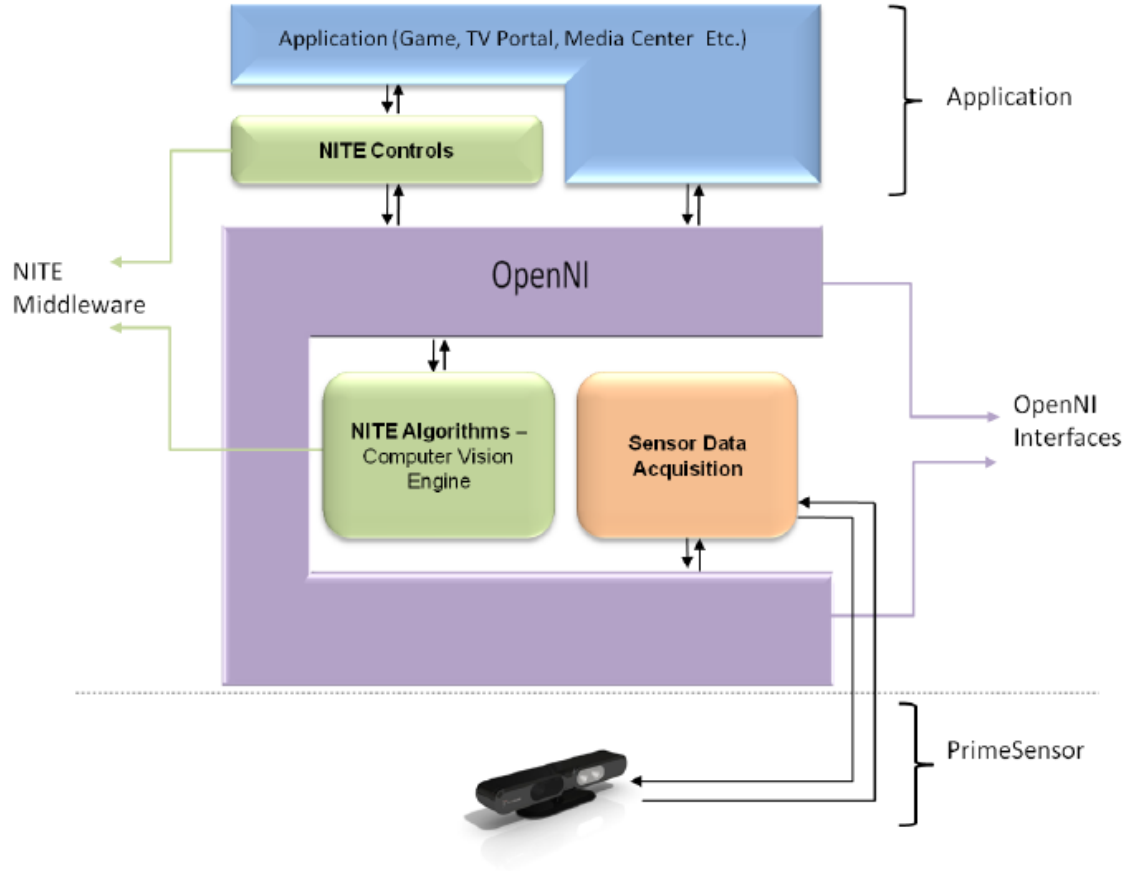


Figura 3: Arquitectura en capas de la adquisición y procesado de los datos de profundidad (extraído de [6]).

En la figura 4, se muestra el diagrama de bloques que sintetiza el algoritmo desarrollado para la creación de la biblioteca con el fin de alcanzar el objetivo propuesto. Como se puede ver, la biblioteca se basa en dos procesos: el principal, encargado de identificar y rastrear el esqueleto corporal de los usuarios, mostrándolos por pantalla; y el secundario, encargado de la creación de la ventana de visualización de los datos que, como su nombre indica, será por donde se visualicen los datos adquiridos por la cámara.

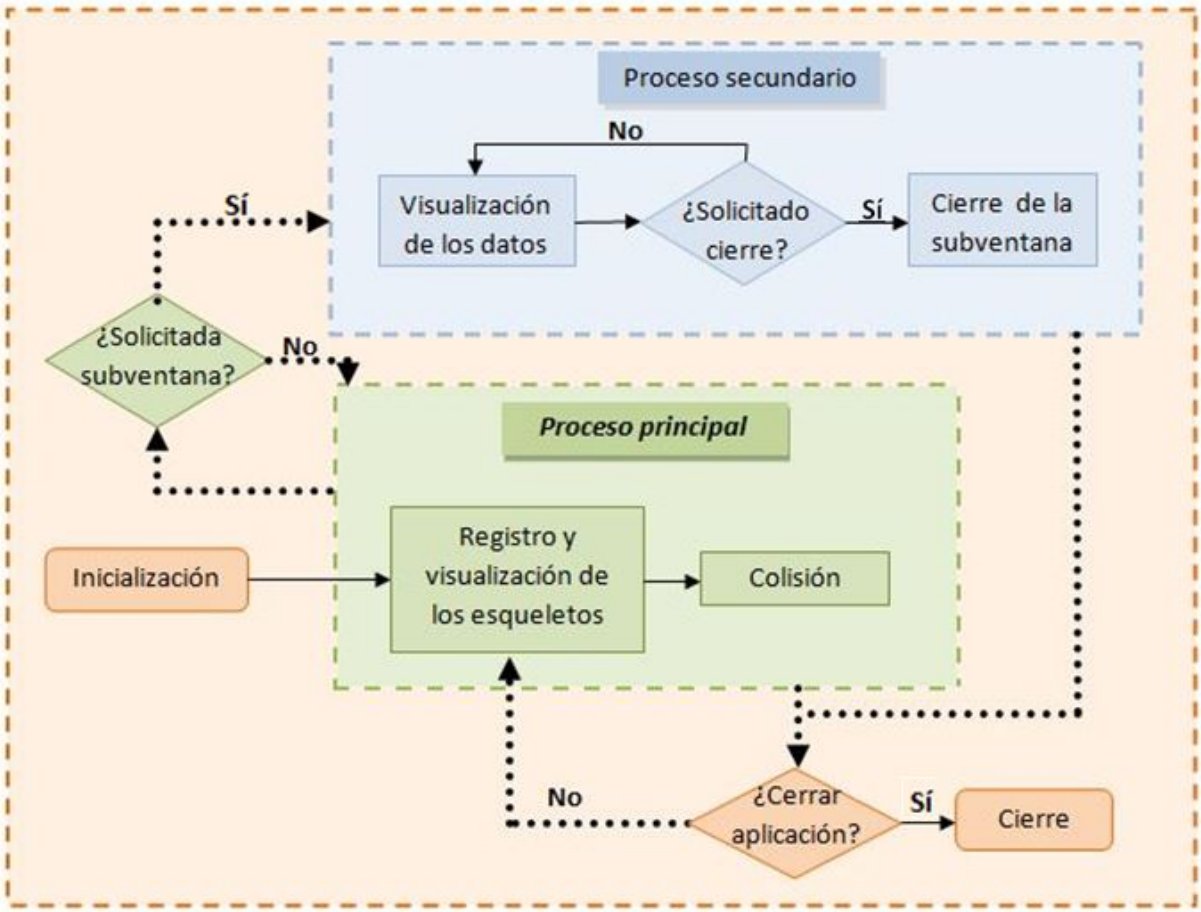


Figura 4: Diagrama de bloques general del algoritmo desarrollado.

Como se puede ver en la figura 4, las funciones del algoritmo se agrupan bajo cinco grandes grupos. A continuación se describe la funcionalidad básica de cada uno de estos bloques.

- *Inicialización.* Etapa encargada de inicializar las librerías mencionadas con anterioridad, así como de calibrar la cámara con la configuración deseada para la ejecución del proyecto.
- *Visualización de los datos.* Etapa encargada del control de la ventana secundaria, asociada a la ventana de OpenGL principal, que mostrará los datos proporcionados por la Kinect.
- *Registro y visualización de los esqueletos.* En esta etapa se realiza la identificación de los usuarios visibles por la kinect y se procede a iniciar el seguimiento de sus esqueletos corporales, esqueletos que se mostrarán a través del entorno virtual.
- *Colisión.* Etapa encargada de verificar si se produce interacción entre el usuario y el entorno virtual, así como de actuar en concordancia cuando se dé esta interacción.
- *Cierre.* En esta etapa se procede a la liberación de la memoria y de los recursos requeridos para el correcto funcionamiento de la aplicación desarrollada. Se divide en dos partes: el cierre de la aplicación y el cierre de la subventana.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Lista de acrónimos

3D	tridimensional.
API	Application Programming Interface.
fps	frames per second.
GEINTRA	Grupo de ingeniería electrónica aplicada a Espacios Inteligentes y Transporte.
MTH	matrices de transformación homogénea.
NiTE	Natural Interaction Technology for End-user.
OpenGL	Open Graphics Library.
OpenNI	Open Natural Interaction.
RGB	Red, Green, Blue.
SoC	System on Chip.
SR	sistema de referencia.
TFG	trabajo de fin de grado.
ToF	Time of Flight.

Capítulo 1

Introducción

Las que conducen y cambian el mundo no son las máquinas, sino las ideas.

Victor Hugo

1.1 Introducción

Desde el inicio, el hombre ha manejado las máquinas que construía de una manera muy directa, centrándose en el contacto físico, mediante el uso de interruptores y palancas, como medio de comunicación entre estas y sus usuarios. Una práctica que sigue manteniéndose actualmente, a través del uso de los distintos periféricos existentes, como el ratón del ordenador. Aunque en la actualidad, una nueva tendencia, basada en los avances tecnológicos que se han sucedido, ha empezado a surgir con el objetivo de desligar al usuario de la necesidad de esa interacción física. El control de los sistemas mediante gestos es uno de los ejemplos que muestran esta nueva pauta de desarrollo y que cada vez adquiere más presencia en nuestra sociedad.

Uno de los sistemas sensoriales en auge son los denominados sensores de profundidad que proporcionan información de distancia entre la cámara y la escena. Este tipo de sensores tiene múltiples aplicaciones en diferentes ámbitos, tales como la robótica móvil [14] o la detección de personas [15].

Entre este tipo de sensores se encuentran las cámaras de ToF [8] en las que la distancia se obtiene de forma indirecta en función del tiempo de vuelo de una señal infrarroja modulada. Las cámaras ToF permiten obtener, además de una imagen de nivel de gris, una imagen de distancias invariante a la iluminación, siendo además la distancia mínima que puede medir bastante inferior al caso de las cámaras basadas en luz estructurada. Sin embargo, aún presentan problemas de precisión por los efectos de la interferencia por multicamino [16] o los artefactos de movimiento [17].

Dadas las limitaciones de las cámaras ToF, en los últimos años han aparecido múltiples trabajos en los que se fusiona la información de ToF con las imágenes adquiridas por una cámara Red, Green, Blue (RGB), para mejorar la calidad de la información disponible. Esta combinación ha sido empleada con éxito en múltiples aplicaciones como el reconocimiento de gestos [18], o la imagen médica [19].

Es en este contexto donde se enmarca este proyecto, que mediante el uso de un cámara Kinect v2, pretende integrar la detección de los usuarios presentes en un espacio físico en un entorno virtual, de tal forma que mediante el seguimiento de los esqueletos corporales de estos usuarios, sea posible la

visualización y el seguimiento de su movimiento en tiempo real dentro de este entorno, permitiendo así mismo una interacción directa del usuario con algunos elementos virtuales.

Este proyecto continúa la línea de trabajo iniciada en el proyecto de Sergio Torrado López [1] y se apoya del proyecto desarrollado por David Casillas Pérez en [2], del cual se extrae el entorno virtual mencionado con anterioridad.

1.2 Objetivos

El objetivo principal de este TFG es la integración de la estimación de los esqueletos de uno o varios usuarios, adquiridos mediante el uso de una cámara ToF, en un espacio virtual inteligente, permitiendo la interacción de estos usuarios con ciertos elementos del espacio a través de sus propios movimientos corporales.

Para la consecución de este objetivo, se ha desarrollado un algoritmo, que como se muestra en la figura 1.1, se compone de un proceso principal y un proceso secundario, que engloban en sí mismo cinco grandes funciones, que se explicarán brevemente a continuación. Estos procesos y sus funciones asociadas se explican en mayor detalle en el capítulo 3.

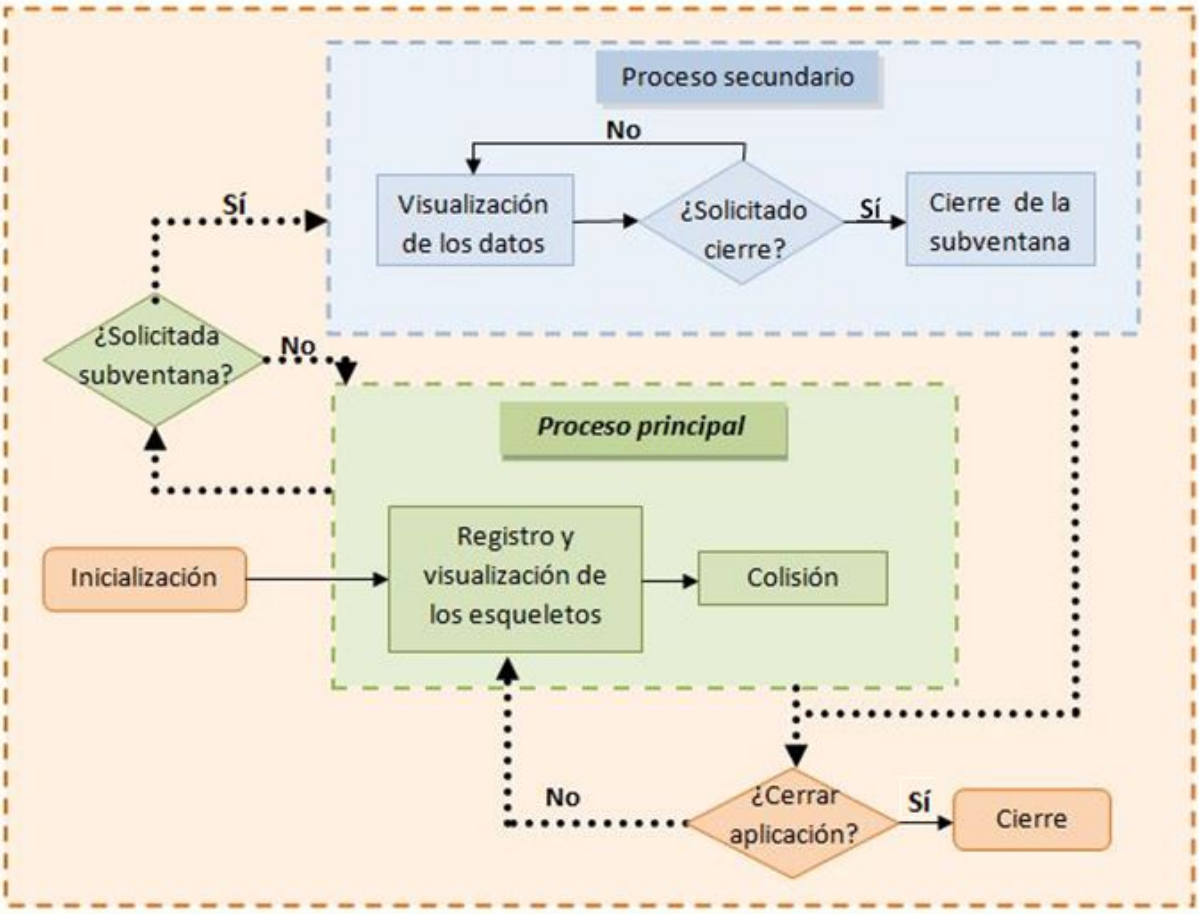


Figura 1.1: Diagrama de bloques general del algoritmo desarrollado.

Como se ha mencionado con anterioridad, en la figura 1.1 se muestran los bloques funcionales que componen el algoritmo desarrollado, y de los cuáles se proporciona una breve descripción a continuación:

- *Inicialización.* En esta etapa se preparan los sistemas para garantizar el correcto funcionamiento del algoritmo, mediante la inicialización de los middleware OpenNI y NiTE, así como la calibración de la cámara con la configuración deseada para la ejecución del proyecto.
- *Visualización de los datos.* Esta etapa es la encargada de la generación de la ventana de visualización para los datos proporcionados por la Kinect. También se engloban en ella, todas aquellas funcionalidades encargadas de controlar e interactuar con la ventana previamente mencionada.
- *Registro y visualización de los esqueletos.* Esta es la etapa principal del sistema desarrollado, en ella se identifica a los usuarios presentes, registrando las articulaciones que conforman sus esqueletos y traspasando estos datos al entorno virtual para su visualización por pantalla.
- *Colisión.* Esta etapa es la encargada de proporcionar un sistema de interacción con el entorno virtual a los usuarios presentes, en este caso mediante la interceptación de la ruta de movimiento de una pelota por parte del usuario. Para ello, se toman los datos de los esqueletos obtenidos en la etapa anterior y se verifica si su posición geométrica coincide con la de la pelota en ese instante.
- *Cierre.* Esta etapa es la encargada de la liberación de todos los recursos requeridos por la aplicación durante su ejecución, y se encuentra subdividida en dos bloques: el primero, *Cierre de la subventana*, se encarga exclusivamente del cierre de la ventana de visualización de los datos, y el segundo, *Cierre* es el principal, que llamaría al primer bloque en caso de ser necesario.

1.3 Organización de la memoria

El documento que va a leer a continuación y que describe el desarrollo del TFG previamente mencionado, se ha organizado en cinco grandes capítulos.

1. *Introducción.* Exposición del desarrollo en la interacción entre el usuario y la máquina, y su evolución reciente hacia mecanismos que permitan una interacción sin contacto físico, que constituye la causa principal del estudio desarrollado en este TFG. Así como explica la importancia de las cámaras ToF empleadas.
2. *Estudio teórico.* A lo largo de este capítulo se realiza un estudio de los diferentes componentes involucrados, principalmente las características de funcionamiento y tipos de cámaras ToF, los middlewares integrados en el desarrollo y el análisis matemático involucrado en la definición del algoritmo.
3. *Desarrollo.* En este capítulo se explica con mayor detalle los procesos que conforman el algoritmo del sistema, así como los distintos bloques integrados en ellos.
4. *Resultados.* En este capítulo se muestran y evalúan los resultados obtenidos con la ejecución del algoritmo desarrollado.
5. *Conclusiones y líneas futuras.* En este capítulo se concretan las conclusiones alcanzadas tras observar los resultados obtenidos y se proponen posibles líneas de mejora y diversificación respecto a lo desarrollado.

Capítulo 2

Estudio teórico

Todas las verdades son fáciles de entender una vez descubiertas; el punto es descubrirlas.

Galileo Galilei

2.1 Introducción

En este capítulo se exponen las bases sobre las que se sustenta este TFG. Así, en los siguientes apartados se detallan los tipos de cámaras de profundidad que se pueden obtener en el mercado, en base a su tecnología de medida y concentrándose fundamentalmente en las que hacen uso de la tecnología ToF, ya que es el tipo que se ha empleado en la realización de este trabajo. A continuación, se analizan los componentes software empleados, describiendo su funcionalidad básica y aquellas de las funciones de las que se componen de las que se ha hecho uso. Finalmente, se realiza un análisis de los componentes matemáticos que influencian este trabajo y su desarrollo, siendo estos el traspaso de coordenadas entre sistemas de referencia y la intersección de figuras geométricas en el espacio tridimensional (3D).

2.2 Cámaras de profundidad

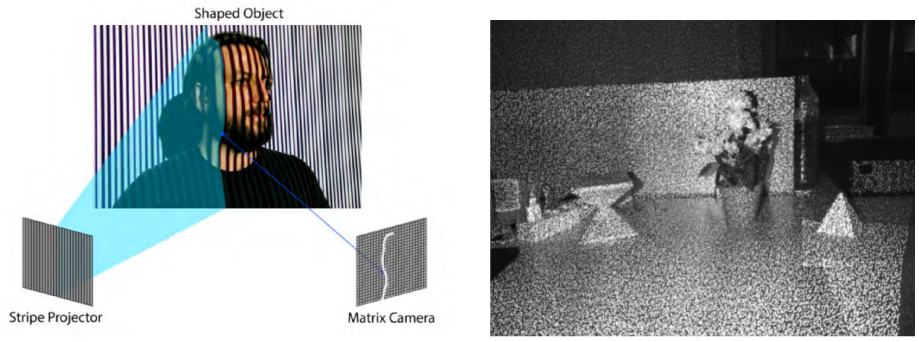
En la actualidad, las cámaras disponibles en el mercado pueden agruparse bajo 3 grandes grupos en función del tipo de imagen que puedan captar, siendo estas imágenes de color, de profundidad o infrarrojas. La Kinect v2 que se ha empleado en la realización de este proyecto, es una cámara que presenta las 3 opciones, al componerse de una cámara RGB, que permite la captación de imágenes de color y, un emisor infrarrojo junto a un sensor de profundidad infrarrojo, siendo este par emisor-receptor el que permite la generación de imágenes infrarrojas y de profundidad, el emisor emite un haz de luz modulada que se refleja sobre los objetos presentes, mientras que el receptor capta esta reflexión y la convierte a información de profundidad.

Las ventajas principales de las imágenes de profundidad, respecto a las otras dos opciones, residen en que los sensores de profundidad son invariantes a la luz ambiental y que, al contrario que en los otros dos casos, este tipo de imágenes no proporciona datos que puedan facilitar la identificación de las personas captadas, permitiendo preservar su identidad. Es por estos motivos que en este TFG se ha optado por el uso exclusivo de la cámara de profundidad de la Kinect v2.

Dentro de las cámaras de profundidad, se pueden distinguir múltiples tipos en base a la técnica en la que se apoyen para obtener las medidas de profundidad. Entre ellas, dos de las más conocidas son las cámaras basadas en luz estructurada y las basadas en ToF, métodos que se van a describir con más detalles a continuación [7, 20].

2.2.1 Cámaras de profundidad de luz estructurada

La técnica de luz estructurada, utilizada en cámaras como la Kinect v1 de Microsoft [21, 22], se basa en la proyección de un patrón conocido sobre la escena, como el de la figura 2.1a, para su posterior lectura mediante una cámara ubicada en una posición diferente. Mediante el estudio de las deformaciones producidas en el patrón, debido a la presencia de diferentes obstáculos, se obtiene la distancia existente entre el objeto y la cámara, que es lo que se entiende por profundidad.



(a) Técnica de luz estructurada (b) Patrón de luz emitido

Figura 2.1: Luz estructurada: técnica y patrón (extraídas de [7]

Los sensores que aplican esta técnica proyectan un patrón de motas conocido, como el de la figura 2.1b, a través de sus emisores infrarrojos. Patrón que es posteriormente captado por la cámara infrarroja que poseen estos sensores para, a través del análisis de las variaciones de intensidad producidas, determinar la profundidad existente hasta el punto en cuestión. Este análisis se conoce con el nombre de interferometría de moteado. En la figura 2.2 se muestra este proceso, donde primero se obtiene el infrarrojo, a continuación se estima la disparidad y finalmente, se obtiene la imagen de profundidad.

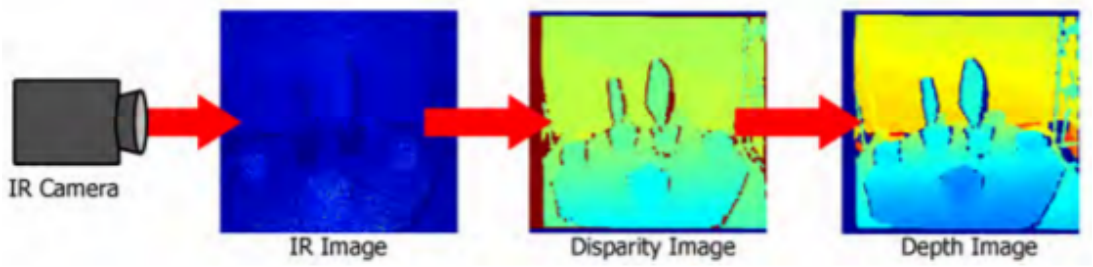


Figura 2.2: Proceso de adquisición de la imagen de profundidad (extraído de [7]).

La relación entre esa disparidad y la profundidad se muestra en la figura 2.3, donde para determinar la profundidad Z^k de un punto k de la imagen obtenida, el sensor compara esta imagen con la de un plano de referencia previamente obtenido a una profundidad Z^o, siendo o el punto correspondiente a k en el plano de referencia. Esto provoca un desplazamiento horizontal D en el plano del objeto, que es percibido por la cámara como una disparidad d. De esta forma, conociendo los parámetros intrínsecos

de la cámara, distancia focal (f) y *baseline* o distancia entre el emisor y el receptor de la cámara (b), se puede calcular la profundidad Z^k a la que se encuentra el punto en cuestión, mediante triangulación, en base a las ecuaciones 2.1 y 2.2, extrapoladas de la figura 2.3

$$\frac{D}{b} = \frac{Z^o - Z^k}{Z^o} \tag{2.1}$$

$$\frac{d}{f} = \frac{D}{Z^k} \tag{2.2}$$

donde D es el desplazamiento horizontal, b es la *baseline*, f es la distancia focal, d es la disparidad, Z^o es la distancia al punto en el plano de referencia y Z^k es la distancia al punto en la imagen obtenida. Sustituyendo el valor de D de la ecuación 2.1 en la ecuación 2.2, se obtiene la ecuación 2.3 para el cálculo de la distancia Z^k.

$$Z^k = \frac{Z^o}{1 + \frac{d}{bf} Z^o} \tag{2.3}$$

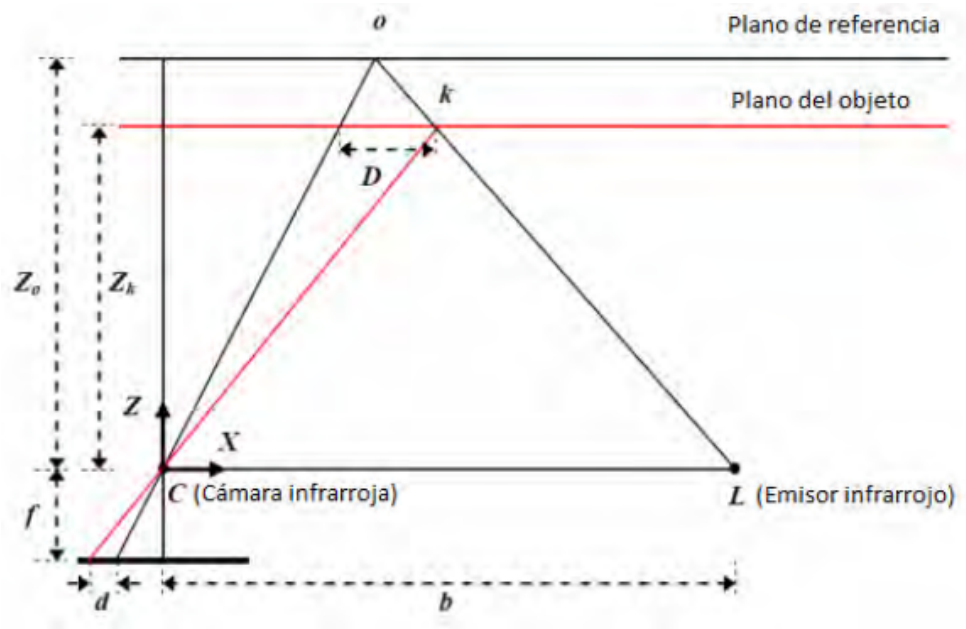


Figura 2.3: Relación entre la profundidad y la disparidad obtenida (extraído de [7]).

Este tipo de cámaras presentan un error de desplazamiento entre la imagen infrarroja y la de profundidad, del orden de 2 a 4 píxeles, que se puede corregir fácilmente obteniendo ambas imágenes, de una escena en la que se distingan claramente los bordes de alguno de los objetos presentes. Una vez obtenidas, se calcula la media de desplazamiento entre los píxeles de ambas imágenes.

Además de este error, algunas de las limitaciones de este tipo de cámaras son el rango de distancias, la dependencia a la iluminación debido a su necesidad de ver el patrón proyectado o el ruido introducido por oclusiones debidas a la diferencia en posicionamiento del emisor y el receptor de la cámara infrarroja, que provocan que cada uno de ellos posea un campo de visión diferente.

2.2.2 Cámaras de profundidad ToF

La tecnología ToF [8] se basa en la utilización de un emisor, normalmente infrarrojo para hacer a la cámara tan invariable a la luz ambiental como sea posible, junto a su correspondiente receptor para calcular, de forma indirecta y en función del tiempo, la distancia a la que se haya un determinado objeto.

Como se ve en la figura 2.4, el emisor envía un haz de luz modulada a una frecuencia f conocida, que rebotará sobre los diferentes objetos presentes en el área, provocando el retorno de la señal hasta el receptor de la cámara. Estas dos ondas presentan una diferencia de fase, debida al tiempo de vuelo de la señal, cuya medición permite calcular la distancia a la que se encuentra el objeto.

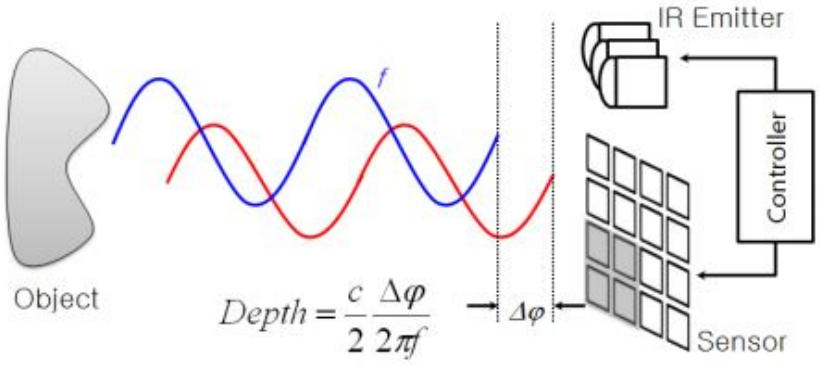


Figura 2.4: Principio de funcionamiento de un sensor ToF (extraído de [8]).

La diferencia de fase, t_d, previamente mencionada, se calcula, para cada píxel, en base a la relación existente entre los 4 valores de carga eléctrica (Q1-Q4) del sensor, de la forma indicada en la ecuación 2.4. Estos valores se calculan mediante la comparación de la señal recibida respecto a 4 señales de control (C1-C4), cada una de ellas desfasada 90º grados respecto a la señal emitida, y determinan la cantidad de electrones recolectados de la señal reflejada. Este funcionamiento se refleja en la figura 2.5.

$$t_d = arctan(\frac{Q3 - Q4}{Q1 - Q2}) \tag{2.4}$$

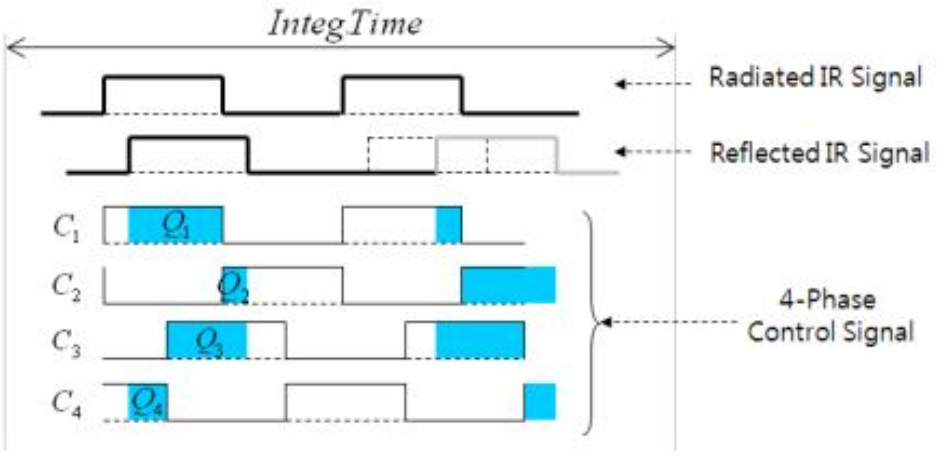


Figura 2.5: Diagrama de funcionamiento de un sensor ToF (extraído de [8]).

Una vez conocida la diferencia de fase de la señal, se puede determinar la distancia d a la que se encuentra el objeto, como se muestra en la ecuación siguiente.

$$d = \frac{c}{2f}\frac{t_d}{2\pi} \tag{2.5}$$

donde c es la velocidad de la luz y f la frecuencia de la señal emitida.

Esta misma ecuación (2.5), nos permite determinar la distancia máxima de medida (d_{max}) de una cámara ToF para que esta no presente ambigüedad, siendo esta la mencionada en la ecuación 2.6. Esto es debido a que la diferencia de fase, t_d, debe estar comprendida entre 0 y 2π radianes, por lo que cualquier valor superior a este, generará únicamente una repetición de lo ya obtenido, al tratarse de una señal periódica que se repite cada 2π, y generará ambigüedad al ser imposible distinguir si se trata de una diferencia de fase β o $\beta + 2\pi$.

$$d_{max} = \frac{c}{2f} \tag{2.6}$$

Al igual que con la técnica de luz estructurada, la tecnología ToF no esta exenta de limitaciones y errores [8, 20], fallos que afectan a la precisión y fiabilidad de las medidas tomadas. En el caso de las cámaras ToF, como la kinect v2, estos errores se pueden clasificar en 2 grandes grupos: errores sistemáticos y errores no sistemáticos, que se definen con más detalle a continuación.

- **Errores sistemáticos.**
  Este tipo de errores suele aparecer en la transformación de la luz recibida a señal y tienden a ser compensados vía hardware, los más destacables son:
  - *Error de demodulación infrarroja*: es el error que se produce en la medida de la profundidad debido a la necesidad de aproximar la señal recibida a un tipo de señal conocido, como una sinusoidal, antes de realizar su comparación con la señal de referencia.
  - *Error de tiempo de integración*: un tiempo más largo de integración proporciona mayor precisión en la medida al amplificar la relación de la señal respecto al ruido pero, al mismo tiempo, ralentiza la toma de imágenes.
  - *Variación en la amplitud*: la amplitud de la señal reflejada depende de la reflectividad y el color del objeto, así como de su distancia a la cámara. Estos factores introducen ruido que afecta al cálculo de la profundidad.
  - *Errores debidos a la temperatura*: producidos por la propia temperatura de la cámara, las cámaras ToF se ven afectadas por este tipo de error debido a la potencia necesaria para generar el patrón de luz adecuado para alcanzar una relación señal-ruido (SNR) lo suficientemente óptima.
- **Errores no sistemáticos.**
  Aquellos errores que no se pueden predecir pero afectan a la medida, entre ellos destacan:
  - *Dispersión de la luz*: objetos cercanos a la cámara provocan saturación infrarroja, que debido a la baja sensibilidad de la cámara, generan distorsiones en la lectura de profundidad, obteniendo lecturas que sitúen a los objetos más cerca de los que están en realidad.
  - *Error por multicamino*: ocurre cuando el cálculo de la profundidad para un determinado píxel depende de una señal recibida compuesta de la superposición de múltiples haces infrarrojos reflejados, este error se nota principalmente sobre las áreas cóncavas de los objetos.

– *Ambigüedad de los bordes*: en los bordes de los objetos se generan áreas en las cuáles parte de los píxeles forman parte del objeto, mientras que otros pertenecen a áreas más lejanas. Este error es especialmente importante cuando se requiere realizar la reconstrucción de una escena 3D.

– *Error debido al movimiento*: uno de los errores más importantes que se produce, especialmente cuando se requiere captar los datos para el seguimiento en tiempo real o reconstrucción de la escena 3D. Se produce cuando el objeto en cuestión se mueve de posición durante el tiempo de procesamiento de la escena de la cámara, generando errores en la medida.

2.2.3 Cámara Kinect v2

Como se ha mencionado a lo largo de este capítulo, la Kinect v2 [9, 10, 23], desarrollada por Microsoft, es una cámara ToF compuesta por un sensor de color, un emisor de infrarrojos y un sensor infrarrojo. Esta combinación de sensores le permiten a esta cámara obtener 3 tipos diferentes de imágenes: imagen RGB, imagen de infrarrojos e imagen de profundidad, se puede ver un ejemplo de cada una de estas imágenes en la figura 2.7. Además de estos sensores, Kinect también posee un conjunto de 4 micrófonos, convirtiendo a esta cámara en una de vídeo y audio, lo que la provee de una gran flexibilidad. En la figura 2.6 se muestra una kinect v2 con estos componentes señalizados.

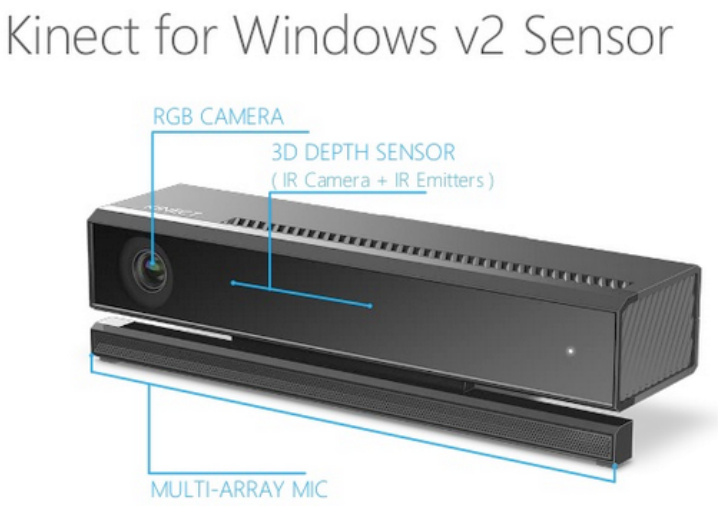


Figura 2.6: Cámara Kinect v2.

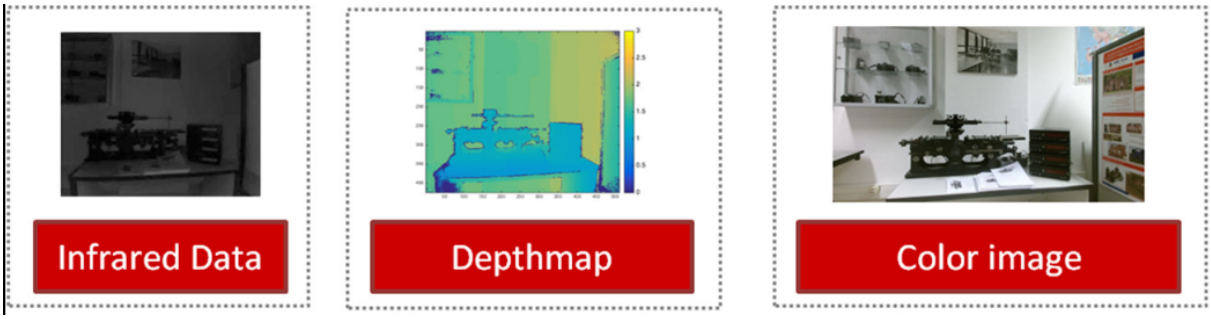
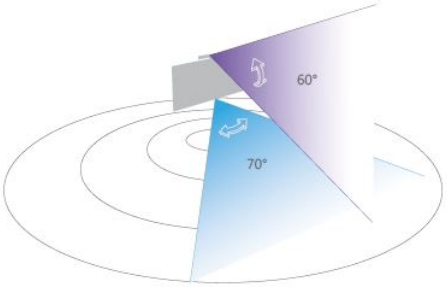


Figura 2.7: Ejemplo (extraído de [3]) de los tres tipos de imágenes obtenibles mediante el uso de una Kinect v2. A la izquierda la imagen de infrarrojos, en el centro la imagen de profundidad y a la derecha la imagen de color.

Las características principales de la kinect v2 son:

- Ángulos de visión: 70º en horizontal y 60º en vertical, de la forma mostrada en la figura 2.8a.
- Apertura focal F<1.1
- Resolución de profundidad dentro del 1 % de la distancia medida.
- Medidas de profundidad válidas entre 0.5m y 4.5m, de la forma mostrada en la figura 2.8b.
- Imágenes de profundidad e infrarrojos de 512x424 píxeles.
- Independiente de la iluminación ambiental.
- Tiempo máximo de exposición de 14 ms.
- Error de la medida menor al 2 % dentro del rango de operaciones.
- Comunicación mediante un puerto USB 3.0 con una latencia inferior a 20 ms.
- Detección y seguimiento de 6 personas, identificando 25 articulaciones por cada una.



(a) Ángulos del campo de visión de una kinect v2

(b) Distancia visible por una kinect v2

Figura 2.8: Características del campo de visión de una Kinect v2 (imágenes extraídas de [9]

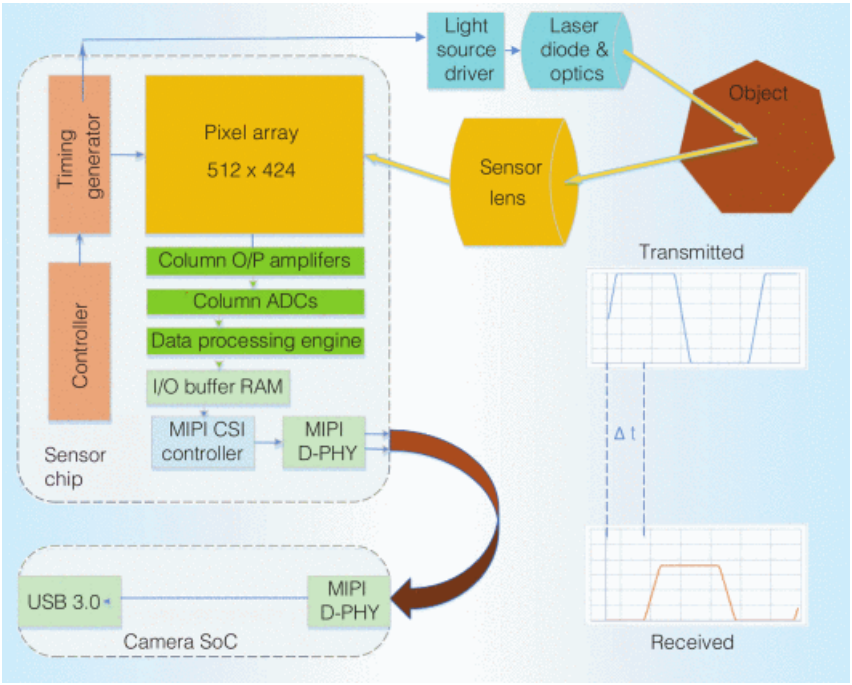


Figura 2.9: Arquitectura interna de la cámara kinect v2 (obtenida de [10]).

En la figura 2.9, se puede apreciar la arquitectura de bloques interna de esta cámara, que como se puede ver, se compone de: el chip del sensor, una cámara System on Chip (SoC), el sistema de iluminación y los sensores ópticos para la generación de la imagen.

Como una cámara basada en la tecnología ToF, sobre la que se ha hablado en la sección anterior (2.2.2), la kinect v2 también se ve sometida a los errores generados por este tipo de técnica, algunos de estos errores son: la variación de la amplitud, los errores debidos a la temperatura, la ambigüedad presente en los bordes de los objetos y el error debido al movimiento durante la fase de procesado de imágenes de la cámara; errores explicados brevemente en la sección anterior.

2.3 Análisis de recursos software utilizados

En esta sección se hace referencia a los distintos componentes de software de los que se ha hecho uso para el desarrollo del algoritmo, siendo estos principalmente: OpenNI, que proporciona una API que permite la interacción entre dispositivos hardware y los middlewares que interactúen con ellos; NiTE, un middleware de percepción que interpreta los datos de dispositivos hardware para generar distintas funcionalidades, como el rastreo del esqueleto; y, OpenGL, una API que permite la generación de gráficos 3D.

2.3.1 OpenNI

OpenNI [4,24] es una organización sin ánimo de lucro, creada en Noviembre de 2010, para la certificación y mejora de la interoperatibilidad de la interfaz natural de usuario y la interfaz orgánica de usuario para dispositivos de interacción natural, las aplicaciones que requieren del uso de estos dispositivos y el middleware que interactúa con ellos. Uno de sus miembros principales era PrimeSense, empresa creadora de la tecnología utilizada en la Kinect de Microsoft, por lo que, tras su adquisición en Abril de 2014 por Apple, el sitio web oficial de OpenNI, *OpenNI.org*, fue cerrado. Tras el cierre, las organizaciones que utilizaban esta tecnología conservaron la documentación y los binarios en la página Structure.io [25].

Como se puede ver en la figura 2.10, donde se muestra el proceso seguido con los datos de profundidad desde el hardware hasta la aplicación que hace uso de ellos, OpenNI proporciona una API que permite la interacción de los dispositivos hardware compatibles, ya sean de audio, video o profundidad, con las aplicaciones, permitiéndoles a estas últimas acceder a los datos adquiridos por estos dispositivos. Al mismo tiempo, OpenNI proporciona una interfaz sobre la que otros middlewares se pueden apoyar para interactuar con los dispositivos hardware, simplificando de esta forma su acceso a ellos y amplificando su funcionalidad mediante la integración de procesos como la detección de gestos.

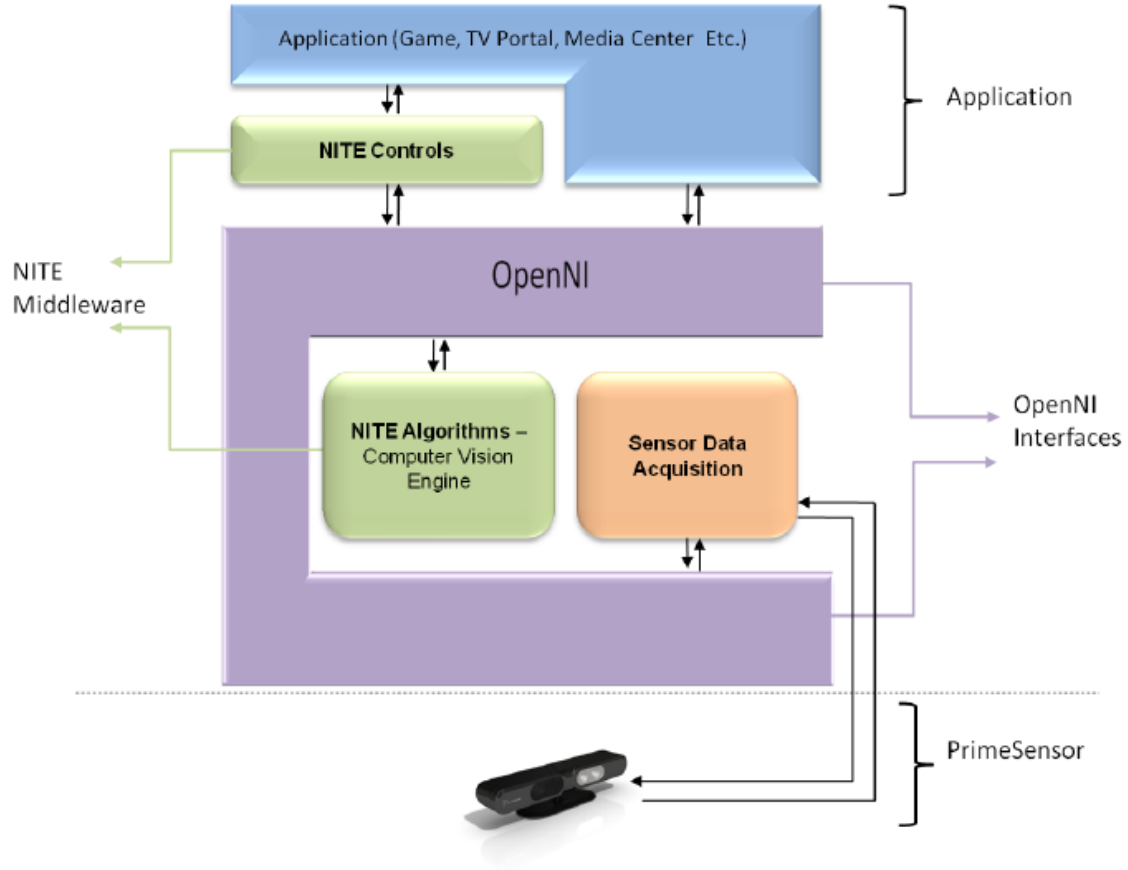


Figura 2.10: Arquitectura en capas de la adquisición y procesado de los datos de profundidad (extraído de [6]

Para la realización de este proyecto se ha empleado OpenNI 2. La API proporcionada se subdivide en 4 grandes clases, siendo estas: *OpenNI*, *Device*, *VideoStream* y *VideoFrameRef*, que en breve se describirán con más detalle. Además de estas 4 clases principales, otras clases y estructuras, como la clase *VideoMode* o la estructura *RGB888Pixel*, son proporcionadas para cubrir el manejo de ciertos tipos específicos de datos.

A continuación, se hace una breve introducción de las clases principales de OpenNI junto a aquellas de las que se ha hecho uso durante el desarrollo de este proyecto, siendo estas:

- **OpenNI**
  Esta clase es la encargada de proveer un punto de entrada para la API mediante la función de inicialización, función que habilita los drivers y localiza los dispositivos conectados, por lo que, para poder acceder al resto de funcionalidades es necesario haberla llamado con anterioridad. Del mismo modo, también se encarga de realizar el cierre de los sistemas mediante la función de apagado. Mientras que está clase se haya inicializado, se podrán crear objetos asociados a los dispositivos y acceder a los datos proporcionados por estos.
  Otras de sus funcionalidades son:
  - Proporcionar la información sobre la versión.
  - Proporcionar la información de errores producidos.
  - Proveer acceso a la información adquirida por los dispositivos, este acceso se puede dar de forma única, esperando hasta que haya datos disponibles, o por eventos.

- **Device**
  Esta clase genera un vínculo con uno de los dispositivos o, en el caso de no existir dispositivo físico, con un dispositivo simulado a partir de los datos proporcionados por un archivo tipo ONI. Su principal función es la de proveer un flujo de datos a través de la creación del objeto en memoria. Esta clase se usa principalmente para conectar, configurar y desconectar el dispositivo

- **VideoStream**
  Esta clase proporciona un método de control sobre el flujo de datos accesible mediante el uso de *Device*. Sus funcionalidades básicas son:

  - Creación, inicialización y destrucción de la clase *VideoStream*.
  - Lectura de los datos, mediante el acceso a los datos más recientes provistos por la clase *VideoFrameRef*.
  - Obtención de información (tipo y configuración del sensor, parámetros fijados de " cropping" y de "mirroring", entre otros)

- **VideoFrameRef**
  Esta clase almacena los datos asociados a una única imagen de entre todas las que llegan a través del flujo de datos, y proporciona toda la información relativa a ella como su formato (anchura y altura), los parámetros de " cropping" que se hayan fijado sobre ella, o el momento del tiempo en que se ha tomado la imagen.

- **VideoMode**
  Esta clase almacena los valores de resolución, la velocidad de la toma de imágenes de la cámara (frames per second (fps)) y el formato de los píxeles que componen la imagen. *VideoMode* puede usarse de forma directa para fijar la configuración de las características mencionadas o puede ser llamada desde otras clases como *VideoStream*, para fijar o verificar la configuración; o *SensorInfo*, para obtener la lista de modos de configuraciones válidas para el dispositivo en uso.

- **DeviceInfo**
  Esta clase almacena los datos de identificación del dispositivo como su nombre identificativo o el nombre del vendedor.

En la figura 2.11 se muestra una esquematización de todas las funciones de OpenNI empleadas para la realización del algoritmo desarrollado, mostrando la relación de dependencia de unas respecto a las otras. Para una explicación detallada de las funciones mencionadas, así como de aquellas de las que no se ha hecho uso, referirse a [26].

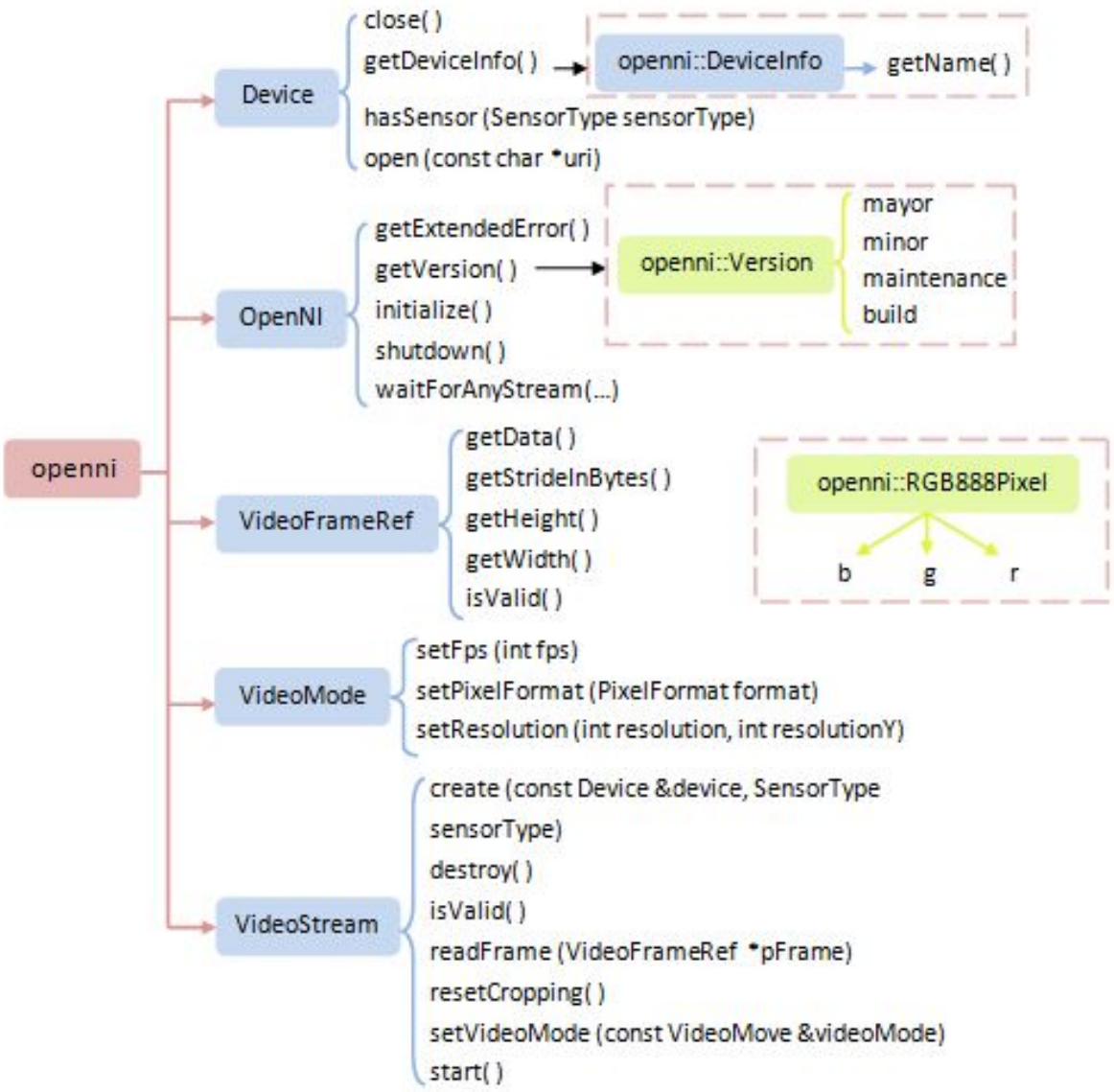


Figura 2.11: Diagrama de dependencia entre las clases de OpenNI, y sus funciones asociadas, en base a su uso en el algoritmo desarrollado. Namespace en rosa, clases en azul y estructuras en amarillo.

2.3.2 NiTE

En la actualidad, el desarrollo de dispositivos basados en interacción natural, es decir, en una interacción hombre-máquina basada en los propios sentidos humanos, principalmente el oído y la vista, se ha vuelto cada vez más común. Un ejemplo de este tipo de dispositivos son las cámaras de profundidad, capaces de interpretar el mundo que las rodea mediante la obtención de los datos de profundidad. Pero estos datos, son datos sin procesar, por lo que es necesario la aplicación de un elemento de procesamiento capaz de interpretarlos y extrapolar de ellos información significativa que pueda ser interpretada por el usuario final como, por ejemplo, la posición de las personas presentes o sus movimientos.

Es bajo esta necesidad que surge NiTE [6], un middleware de percepción 3D que fue lanzado al mercado en Diciembre de 2010 por la empresa PrimeSense. NiTE tiene la capacidad de interpretar los datos percibidos por un dispositivo hardware y convertirlos a datos significativos de una forma similar a como lo haría una persona. Para ello, se compone de algoritmos de visión por computadora, que le permiten identificar a las persones y objetos presentes, detectar y rastrear el movimiento de las manos

y realizar el seguimiento del movimiento del esqueleto corporal; así como, de una API para implementar controles de interfaz de usuario basados en interacción natural, principalmente en gestos, que le permitan al usuario controlar la aplicación desarrollada mediante ellos. En la figura 2.10 del apartado anterior (2.3.1) se puede observar la arquitectura aquí descrita.

Al igual que en el caso de OpenNI, descrito en el apartado anterior, NiTE también se compone de una combinación de clases que agrupan todas las funciones que lo componen. A continuación, se describen aquellas que han sido relevantes para la realización de este proyecto:

- **NiTE**
  Esta clase es la encargada de proporcionar un punto de acceso a la API, lo que logra mediante la función de inicialización, por lo que es imperativo llamar a esta función si se desea utilizar alguna otra de sus funcionalidades de la interfaz. Del mismo modo, también debe llamarse a la función de apagado cuando se desee detener la aplicación, para garantizar el cierre correcto de la interfaz. Por último, esta clase es la encargada de proporcionar los datos de la versión de NiTE que esta siendo utilizada al usuario.

- **Skeleton**
  Esta clase es la principal del algoritmo de control del esqueleto integrado en la API. El propósito principal del algoritmo mencionado, es la identificación de la posición y orientación de las articulaciones del usuario, partiendo de los datos de este que proporciona la clase *UserTracker*. En el caso de que el usuario no sea completamente visible, el algoritmo estimará la posición de aquellas articulaciones fuera del campo de visión de la cámara, estimación que es distinguible del caso presente por el factor de confianza asociado a ella.
  NiTE permite la obtención del esqueleto bajo 2 métodos: la calibración automática, que requerirá de unos segundos para estabilizarse, o la calibración mediante pose, que implica que el usuario debe estar en una pose, previamente determinada, antes de que se pueda iniciar la calibración de su esqueleto.

- **UserData**
  Esta clase proporciona los datos disponibles sobre un determinado usuario, datos proporcionados por la clase *UserTracker*. Las funcionalidades principales de esta clase son:

  - Determinar el estado del usuario: acaba de aparecer, se acaba de marchar o es visible.
  - Obtener los datos del esqueleto del usuario, su identificador o la pose en la que se encuentra.

- **UserTracker**
  Esta es la clase principal de NiTE, su funcionalidad principal es la identificación de los usuarios presentes, distinguiéndolos tanto unos de otros como del fondo. Una vez identificado un usuario, le asigna un identificador que permanecerá constante mientras que el usuario permanezca en el campo de visión y sea visible. Otras funcionalidades de esta clase son:

  - Conversión de coordenadas, del mapa de profundidad al "mundo" o viceversa.
  - Inicializar y detener el rastreo del movimiento o la pose del usuario.
  - Obtener o fijar el factor de suavizado ("smoothing factor") del esqueleto. Este factor controla la velocidad de respuesta del esqueleto ante los movimientos del usuario.

- **UserTrackerFrameRef**
  Esta clase almacena la información relativa a una única imagen de las obtenidas mediante el uso de *UserTracker*. Contiene la información relativa a los usuarios y al suelo.

En la figura 2.12 se muestra una esquematización de todas las funciones de NiTE empleadas para la realización del algoritmo desarrollado, mostrando la relación de dependencia de unas respecto a las otras. Para una explicación detallada de las funciones mencionadas, así como de aquellas de las que no se ha hecho uso, referirse a [26].

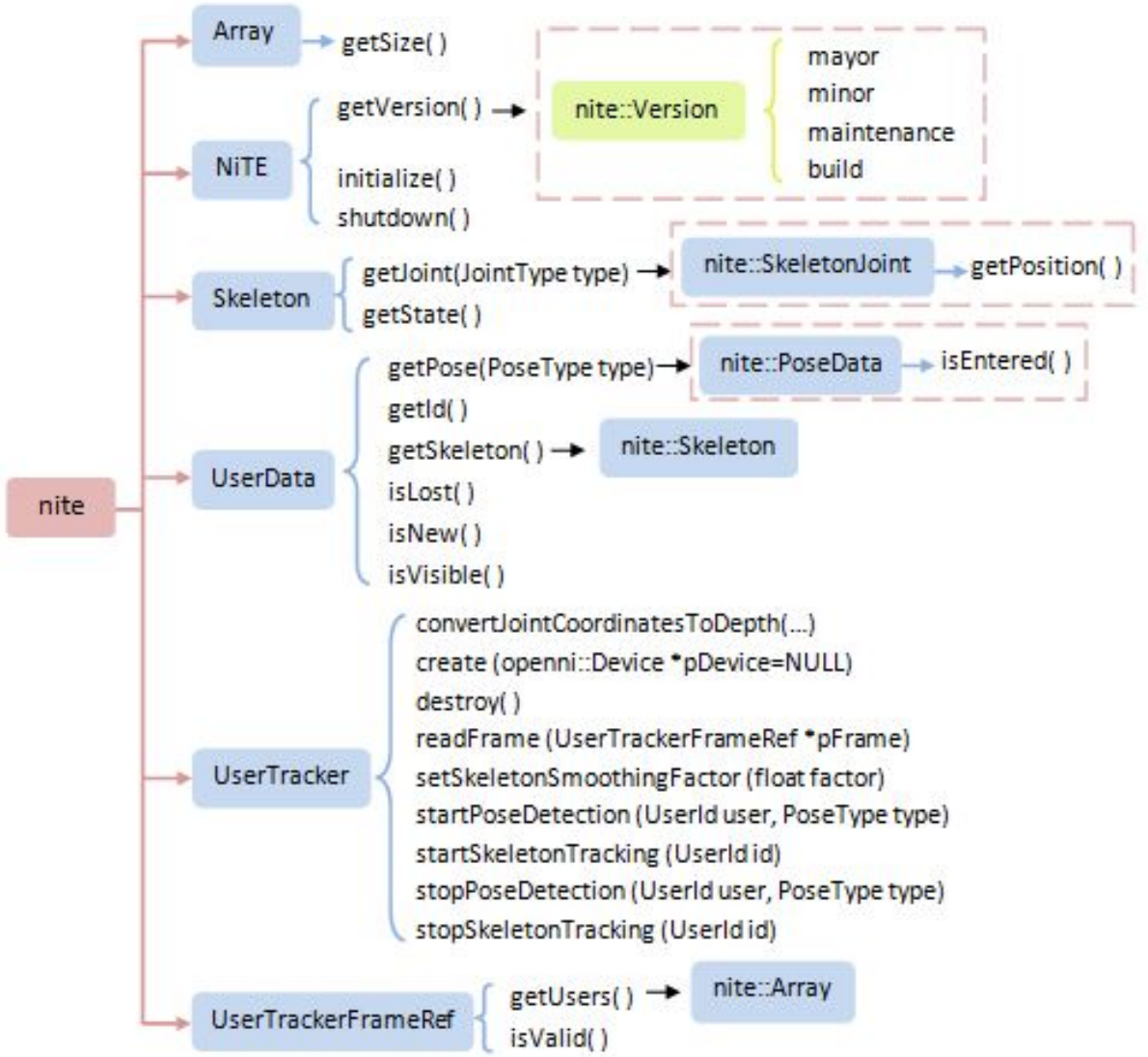


Figura 2.12: Diagrama de dependencia entre las clases de NiTE, y sus funciones asociadas, en base a su uso en el algoritmo desarrollado. Namespace en rosa, clases en azul y estructuras en amarillo.

2.3.3 OpenGL

OpenGL [11] es una API diseñada para la generación y manipulación de gráficos 2D y 3D. Esta librería fomenta la innovación y el desarrollo de aplicaciones gráficas gracias a la gran variedad de funcionalidades que presenta. Entre estas se encuentran la capacidad para incorporar efectos especiales a una escena, la generación de diferentes texturas y la representación de estas.

La librería, cuyo diagrama de funcionamiento se puede ver en la figura 2.13, se ejecuta con nuestro programa independientemente de la capacidad gráfica de la máquina empleada. Así mismo, también esta preparada para ejecutarse sobre cualquier tipo de sistema operativo, como Windows, Linux o Solaris. Es

esta versatilidad lo que convierte a OpenGL en una de las librerías gráficas más utilizadas y extendidas en el mercado actual.

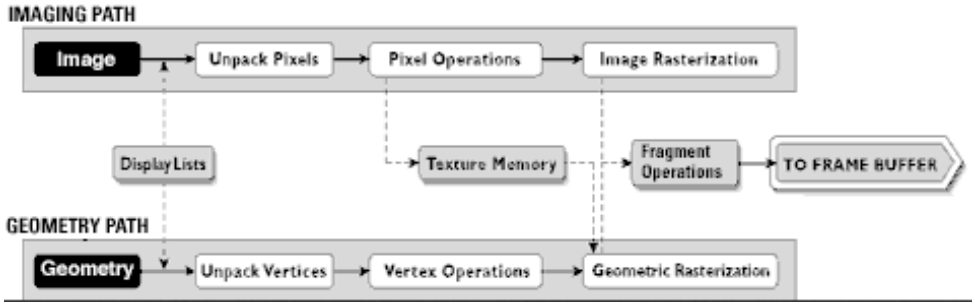


Figura 2.13: Diagrama de funcionamiento de OpenGL (extraído de [11]).

Algunas de los aspectos más importantes de OpenGL y que se han empleado para el desarrollo del algoritmo objetivo de este TFG, son:

- Librería GLUT: esta librería, que se sirve de funciones de OpenGL, agrega funcionalidades como la creación de ventanas o el control de eventos de entrada.

- Texturización: funcionalidad implementada para agregar más realismo a una escena, se basa en la asignación de una imagen a un polígono, de forma que este contenga la imagen en vez de un color plano o un gradiente de colores.

- Buffers: OpenGL dispone de múltiples buffers, siendo los más importantes el de color y el de profundidad. El buffer de color contiene la información del color de los píxeles, cada píxel puede contener un índice de color o valores RGB-Alfa que describan su apariencia. El buffer de profundidad almacena los valores de distancia asociados a cada píxel, siendo esta distancia la existente entre la posición del píxel y la posición del observador.

2.4 Matrices de transformación homogénea

Las matrices de transformación homogénea (MTH) [12] son uno de los métodos de representación espacial existentes. Este método ha sido ampliamente usado en gráficos de computadora para el cálculo de proyecciones y modificaciones en la perspectiva de los objetos representados. Actualmente, también es ampliamente usado en robótica, como medio para el modelado y programación de los robots, mediante la obtención de la relación entre sus diferentes partes.

Las MTH son matrices de 4x4 que representan la transformación de un vector en coordenadas homogéneas de un sistema de coordenadas a otro. En la figura siguiente se muestra un ejemplo de este tipo de transformación.

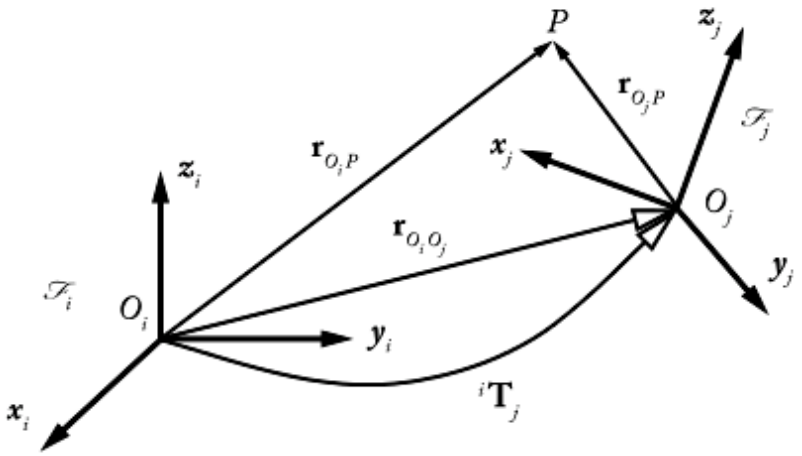


Figura 2.14: Transformación de las coordenadas de un vector de un sistema de referencia a otro (imagen extraída de [12]).

A continuación se procede a explicar cómo obtener la matriz de transformación homogénea existente entre 2 sistemas de referencia. Para entender mejor el cálculo de esta, se tomará como referencia el caso presentado en la figura anterior (2.14) donde, como se puede ver, se tienen 2 sistemas de referencia (F_i y F_j), cada uno de ellos representado por su origen O y sus 3 ejes (x, y, z).

Partiendo de las coordenadas cartesianas del punto P sobre el sistema F_j, se obtienen sus coordenadas homogéneas, que se conforman por 4 componentes, siendo estos: las 3 coordenadas cartesianas multiplicadas por un factor de escalado y el factor de escalado (w), que se suele tomar de valor 1 (si se tratase de un vector el factor de escalado sería 0). De esta forma, el punto P queda representado de la siguiente forma:

$$ {}^j\tilde{p} = \begin{bmatrix} {}^jx_P \\ {}^jy_P \\ {}^jz_P \\ 1 \end{bmatrix} \tag{2.7} $$

Este mismo punto puede ser representado en el sistema F_i, donde sus coordenadas serían ${}^i\tilde{p} = [{}^ix_P, {}^iy_P, {}^iz_P, 1]^T$. Y también, puede ser obtenido como función del punto indicado en la ecuación 2.7, de tal forma que se tendría:

$$ {}^i\tilde{p} = [{}^jx_P\ {}^i\tilde{s}_j + {}^jn_P\ {}^i\tilde{n}_j + {}^jz_P\ {}^i\tilde{a}_j + {}^i\tilde{r}_j] = {}^iT_j\ {}^j\tilde{p} \tag{2.8} $$

donde s, n y a representan las coordenadas en el sistema F_i de los vectores directores unitarios del sistema F_j, y ${}^ir_j$ es el vector representativo en el sistema F_i de las coordenadas del origen O_j del sistema F_j, todo ello en coordenadas homogéneas.

Como se puede ver en la ecuación 2.8, la matriz de transformación homogénea, ${}^iT_j$, permite calcular las coordenadas en el sistema de referencia F_i de un punto del cual se conocen sus coordenadas en el sistema de referencia F_j. Esta matriz define las transformaciones, traslaciones y/o rotaciones necesarias para traspasar un dato de un sistema de referencia al otro y se representa de la forma:

$$ {}^iT_j = [{}^i\tilde{s}_j\ {}^i\tilde{n}_j\ {}^i\tilde{a}_j\ {}^i\tilde{r}_j] = \begin{bmatrix} & {}^iR_j & & {}^ir_j \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9} $$

o, en su forma genérica:

$$T = \begin{bmatrix} s_x & n_x & a_x & r_x \\ s_y & n_y & a_y & r_y \\ s_z & n_z & a_z & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s & n & a & r \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R & r \\ 0\ 0\ 0 & 1 \end{bmatrix} \tag{2.10}$$

donde R representa la matriz de rotación; r, el vector de traslación; $f = (0\ 0\ 0)$, es la transformación de perspectiva; y, $w = 1$, es el factor de escalado.

A la hora de utilizar la MTH se deben tener en cuenta las siguientes propiedades:

1. Para una traslación pura, la matriz de rotación R debe igualarse a la matriz identidad de orden 3; mientras que, para una rotación pura, el vector de traslación r debe ser 0.

2. La matriz de rotación, R, es ortogonal y su determinante es 1, por lo que, su inversa es igual a su traspuesta:
$$R^{-1} = R^T \tag{2.11}$$

3. La inversa de la matriz ${}^iT_j$ es la matriz ${}^jT_i$. Tomando la representación de la matriz dada en la ecuación 2.10, la matriz inversa quedaría:
$$T^{-1} = \begin{bmatrix} & & & -s^T\ r \\ & R^T & & -n^T\ r \\ & & & -a^T\ r \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} & R^T & & -R^T\ r \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.12}$$

4. Se puede verificar fácilmente que:
$$Rot^{-1}(u,\theta) = Rot(u,-\theta) = Rot(-u,\theta) \tag{2.13}$$
$$Tras^{-1}(u,d) = Tras(u,-d) = Tras(-u,d) \tag{2.14}$$

5. La multiplicación de 2 matrices da una nueva matriz de transformación. Esta composición de transformaciones no es conmutativa.
$$T_1\ T_2 \neq\ T_2\ T_1 \tag{2.15}$$

6. Cuando un sistema de referencia F_0, es sometido a k transformaciones sucesivas, y cada una de estas transformaciones i es aplicada sobre el correspondiente F_{i-1}, la matriz de transformación ${}^0T_k$ se puede obtener como producto de las sucesivas transformaciones.
$${}^0T_k = \prod_{i=1}^{k} {}^{i-1}T_i = {}^0T_1 \cdot {}^1T_2 \cdots {}^{k-1}T_k \tag{2.16}$$

7. Cuando se dan transformaciones sucesivas sobre el mismo eje, se cumple:
$$Rot(u,\theta_1)Rot(u,\theta_2) = Rot(u,\theta_1+\theta_2) \tag{2.17}$$
$$Rot(u,\theta)Tras(u,d) = Tras(u,d)Rot(u,\theta) \tag{2.18}$$

2.5 Análisis matemático de las intersecciones en el espacio 3D

En la siguiente sección, se explican los diferentes tipos de intersección en el espacio 3D que se han tenido en cuenta para el desarrollo del algoritmo de colisión, que es el que habilita la posibilidad de interacción entre el usuario y el entorno virtual. Así como se explicará brevemente como se calcula el nuevo vector director del cuerpo en movimiento (la pelota) en el caso de que se produzca colisión.

2.5.1 Intersección línea-línea

En el espacio 3D, la relación que se puede dar entre dos rectas es de 3 tipos distintos, siendo estos:

- Intersección: las rectas pertenecen al mismo plano y comparten un punto común en el espacio.
- Cruce: se da cuando las rectas no pertenecen al mismo plano, es decir, no son paralelas ni pueden interceptarse.
- Paralelismo: las rectas pertenecen al mismo plano pero, nunca llegan a cruzarse.

En este apartado se van a estudiar dos de estos casos: la intersección y el cruce. En concreto, se determinará como obtener el punto de intersección y como calcular la distancia mínima entre 2 líneas sesgadas, o en inglés "skew lines", término con el que se hará referencia a ellas de ahora en adelante.

2.5.1.1 Intersección de 2 líneas en el espacio tridimensional

La condición básica para que dos líneas se intercepten en el espacio es que ambas líneas deben formar parte del mismo plano, es decir, que las líneas deben ser coplanarias.

Para verificar la condición de coplanaridad se pueden aplicar 2 métodos:

1. Si las rectas bajo estudio vienen determinadas por 4 puntos, la coplanaridad se puede determinar mediante la obtención del volumen del tetraedro que forman, que debe ser 0 si los puntos son coplanarias.

$$\begin{vmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{vmatrix} = 0 \tag{2.19}$$

2. Otra forma de verificarlo es aplicar la condición de que las líneas no sean "skew lines". Para que se cumpla esta condición el producto mixto de 3 vectores formados por puntos de estas rectas debe ser 0, como se puede ver en la ecuación 2.20. Los vectores que se definen para el cálculo son los vectores directores de las rectas dadas y un vector que una un punto de cada recta.

$$\vec{a} \cdot [\vec{b} \times \vec{c}] = \begin{vmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{vmatrix} = 0 \tag{2.20}$$

Una vez verificado que ambas rectas forman parte del mismo plano, es necesario descartar la posibilidad de que las rectas sean colineares, es decir, linealmente dependientes la una de la otra, para lo cuál basta con verificar que el producto vectorial de sus vectores directores (V y U) es distinto de 0.

$$\vec{V}_s \times \vec{U}_r = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ s_x & s_y & s_z \\ r_x & r_y & r_z \end{vmatrix} \neq 0 \tag{2.21}$$

Finalmente, habiendo determinado que las rectas se intersecan ya que no son paralelas ni se cruzan, se busca obtener el punto de intersección entre ambas. Para ello, se resolverá el sistema de ecuaciones formado por las ecuaciones de ambas rectas, en este caso se muestra para el caso de las ecuaciones vectoriales de las rectas.

Teniendo 2 rectas s y r, definidas mediante los puntos S_0 y S_F y R_0 y R_F, cuyas ecuaciones vectoriales son:

$$s : P(k) = S_0 + k\ \vec{V}_s = (s_x, s_y, s_z) + k\ (v_x, v_y, v_z) \tag{2.22}$$

$$r : P(l) = R_0 + l\ \vec{U}_r = (r_x, r_y, r_z) + l\ (u_x, u_y, u_z) \tag{2.23}$$

donde P es el punto de intersección, k y l son los parámetros de las rectas y $\vec{V}_s$ y $\vec{U}_r$ son los vectores directores, definidos como $\vec{V}_s = S_F - S_0$ y $\vec{U}_r = R_F - R_0$. Se toma $P(k) = P(l)$ y, bajo esta condición, se resuelve el sistema de ecuaciones para obtener los valores de k y l que cumplan la igualdad mencionada, como se muestra en [27, 28], de forma que los parámetros quedarían de la forma:

$$k = \frac{(\vec{c} \times\ \vec{b}) \cdot\ (\vec{a} \times\ \vec{b})}{|\vec{a} \times\ \vec{b}|^2} \qquad l = \frac{(\vec{c} \times\ \vec{a}) \cdot\ (\vec{a} \times\ \vec{b})}{|\vec{a} \times\ \vec{b}|^2} \tag{2.24}$$

donde $\vec{a} = \vec{V}_s$, $\vec{b} = \vec{U}_r$ y $\vec{c} = R_0 - S_0$.

Una vez obtenidos los parámetros solo hace falta sustituir uno de ellos en su ecuación correspondiente para obtener el punto de intersección. Si se quisiese determinar si los segmentos determinados por los puntos que definen las rectas se intersecan, habría que verificar si los parámetros obtenidos se encuentran entre 0 y 1, de la forma $0 \leq k \leq 1$ y $0 \leq l \leq 1$.

Una vez verificado que se produce colisión, se determina cómo se ve afectado el vector director del cuerpo en movimiento. Tratándose de un caso ideal de choque elástico con un cuerpo rígido inamovible, el ángulo de salida será del mismo valor que el de entrada, respecto al plano definido por la normal a continuación descrita. Para calcular la normal al plano sobre el que este cuerpo incide, ya que solo se conocen dos segmentos, el vector director de la pelota y el segmento que, para este trabajo, representa un hueso del esqueleto, primero se calcula la normal del plano formado por estos dos, y con este y el "hueso- se calcula la normal del plano incidente, de forma que sea unitaria.

$$\vec{N}_{líneas} = \frac{\vec{V}_{hueso} \times\ (\vec{V}_{hueso} \times\ \vec{V}_{pelota})}{|\vec{N}_{líneas}|} \tag{2.25}$$

Obtenida la normal, se calcula el nuevo vector director aplicando el método desarrollado en [13], que se explica en más detalle a continuación.

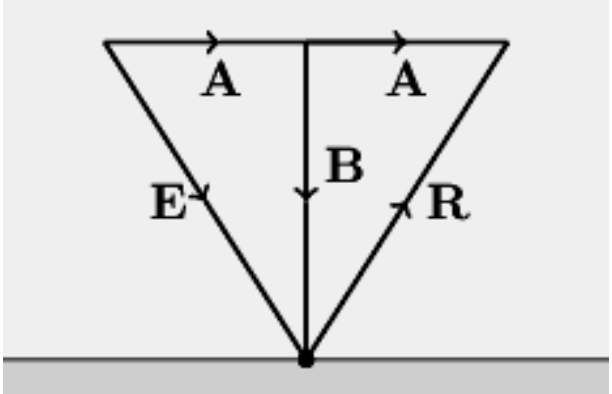


Figura 2.15: Ejemplo de la reflexión de un vector (extraído de [13]).

Tomando el caso de la figura anterior, donde se tiene un vector $\vec{E}$ que incide sobre una superficie con normal $\vec{N}$, reflejándose en la forma del vector $\vec{R}$, se determina que para que esta reflexión se dé, se debe cumplir el siguiente sistema de ecuaciones:

$$\begin{cases} \vec{E} = \vec{A} + \vec{B} \\ \vec{E} = \vec{A} - \vec{B} \end{cases} \tag{2.26}$$

donde los vectores $\vec{A}$ y $\vec{B}$ representan la descomposición del vector $\vec{E}$, de la forma que se observa en la figura 2.15. Ya que el vector $\vec{B}$ es perpendicular a la superficie, su valor queda determinado mediante el producto escalar del vector $\vec{E}$ y la normal de la superficie $\vec{N}$, que se aplicará en su forma unitaria $\hat{N}$.

$$\vec{B} = (\vec{E} \cdot \hat{N}) \cdot \hat{N} \tag{2.27}$$

Sustituyendo la ecuación anterior en el sistema de ecuaciones de 2.26 y despejando $\vec{A}$ de la primera de ellas para sustituirlo en la segunda, se obtiene la ecuación que permite calcular $\vec{R}$, siendo esta:

$$\vec{R} = \vec{E} - 2(\vec{E} \cdot \ \hat{N}) \cdot \ \hat{N} \tag{2.28}$$

2.5.1.2 Cálculo de la distancia mínima entre líneas sesgadas ("skew lines")

Las "skew lines-son rectas no coplanarias, lo que implica que ni son paralelas ni se intersecan, se puede ver un ejemplo de este tipo de rectas en la figura 2.16. Para calcular la mínima distancia existente entre ellas, hay que definir una recta t a partir de un punto de cada una de estas rectas, y buscar con qué punto de las rectas originales la nueva recta t cumple con la condición de perpendicularidad respecto a ambas rectas.

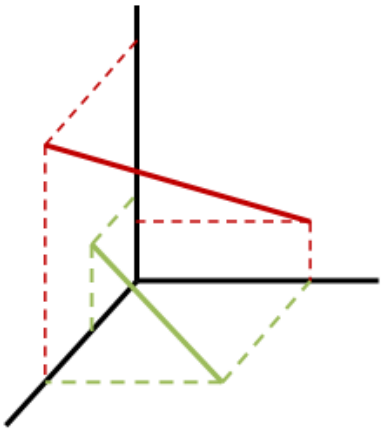


Figura 2.16: Ejemplo de 2 *skew lines*.

Para realizar el cálculo anteriormente descrito, lo primero es verificar que las rectas no sean coplanarias. Como se explico en el apartado anterior (2.5.1.1), para que dos rectas sean de tipo skew se debe cumplir

$$\vec{a} \cdot [\vec{b} \times \vec{c}] = \begin{vmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{vmatrix} \neq 0 \tag{2.29}$$

siendo $\vec{a}$, $\vec{b}$ y $\vec{c}$ tres vectores representativos de las líneas bajo estudio, siendo estos los vectores directores de las rectas y un vector que una un punto de cada una de ellas.

Una vez verificada esta condición, al igual que en el caso de la intersección, habría que definir el sistema de ecuaciones formado por las ecuaciones de las rectas y obtener los parámetros k y l, con los que en este caso se obtendrían los puntos de cada recta de forma que la distancia entre ellas sea mínima [28], que como se ha mencionado antes será cuando la recta que representa la distancia sea perpendicular a ambas rectas. De esta forma las ecuaciones de los parámetros quedarían como se muestra en las ecuaciones 2.24. Y, al igual que antes, si se trabaja con segmentos habría que verificar la pertenencia de los puntos obtenidos a los segmentos correspondientes, para lo que bastaría con verificar que k y l se encontrarán comprendidas entre 0 y 1.

Obtenidos estos parámetros para determinar si se produce colisión debe verificarse si se cumple la condición

$$D = |P(l) - P(k)| \leq R_p \tag{2.30}$$

donde D representa la distancia mínima obtenida y R_p es el radio de la pelota.

Si la condición se cumple se procederá a calcular el cambio en la dirección del vector director de la pelota. En este caso, se aplicará el mismo concepto que en el apartado anterior pero, ya que las rectas nunca llegan a tocarse el cálculo del vector normal que determina esta variación se ve alterado. Para calcular esta normal, lo primero sería determinar la posición de la pelota en la cual su centro P_C se encuentre a una distancia r de la recta definida por los puntos P_0 y P_F. Para ello, se parte del punto del hueso obtenido anteriormente, tomando $P(l)$ como tal, y se calcula el punto P_C del vector director de la pelota $V = P_F - P_0$, que cumpla lo mencionado anteriormente.

$$r = |\vec{d}| = |P_C - P_l| \tag{2.31}$$

donde $P_C = P_0 + tV$. Despejando t de la ecuación anterior se obtiene el punto buscado, ya que se trata de una ecuación de segundo grado, la solución que se busca es aquella en que t este comprendida entre 0 y 1, y el punto resultante sea el más cercano a P_0.

$$t = \frac{-b \pm \sqrt{b_2 - 4ac}}{2a} \tag{2.32}$$

donde $a = r^2 - (x_{P_0-P_l})^2 - (y_{P_0-P_l})^2 - (z_{P_0-P_l})^2$, $b = |\vec{V}|^2$, y $c = 2(x_{P_0-P_l}V_x + y_{P_0-P_l}V_y + z_{P_0-P_l}V_z)$

Tras esto, se calcula el punto de corte Q de la recta que representa el hueso del esqueleto, con el plano perpendicular a esta y que contiene el centro de la pelota P_C, lo que se realiza de la forma descrita en el apartado siguiente (2.5.2), bajo la consideración de que el plano buscado se define mediante el punto P_C y una normal que se corresponde con el vector director de la recta.

Finalmente, con estos datos, se define la normal al plano de colisión, siendo esta el vector de unión del centro de la pelota con el punto Q calculado, de la forma:

$$\vec{N}_{skew} = \frac{P_C - Q}{|\vec{N}_{skew}|} \tag{2.33}$$

Una vez definida la normal, se procederá al cálculo del nuevo vector director de la pelota, que se calcularía aplicando el mismo método descrito para el caso de la intersección (2.5.1.1).

2.5.2 Intersección línea-plano

En el siguiente apartado se describe como determinar si una recta dada corta a un plano definido mediante 3 puntos, para que esto se dé se debe encontrar el punto que satisfaga ambas ecuaciones, la del plano y la de la recta.

Para empezar, se debe obtener la ecuación del plano que contenga a los 3 puntos dados, que quedará definido mediante la normal, $\vec{N}_{plano}$, y uno de los puntos dados. Para obtener la normal se definen 2 vectores con estos 3 puntos, $\vec{V}_1 = P_1 - P_3$ y $\vec{V}_2 = P_2 - P3$, y se calcula su producto vectorial

$$\vec{N}_{plano} = \frac{\vec{V}_1 \times \vec{V}_2}{|\vec{N}_{plano}|} \tag{2.34}$$

Obtenida la normal, faltaría determinar la constante D del plano, para lo cual se sustituyen los valores de la normal y el punto considerado en la ecuación del plano $Ax + By + Cz + D = 0$, de forma que

$$D = -[N_x P_x + N_y P_y + N_z P_z] \tag{2.35}$$

Una vez obtenido el plano, se debe verificar que la recta, cuya intersección se busca, no sea paralela al plano ni este contenida en este. Uno de los métodos para verificar este hecho es la aplicación del producto escalar entre la recta y la normal del plano, siendo esta la obtenida en 2.34.

$$\vec{V}_P \cdot \vec{N}_{plano} = V_x N_x + V_y N_y + V_z N_z \neq 0 \tag{2.36}$$

donde $\vec{V}_P$ es el vector de movimiento de la pelota.

Verificado este punto, se sabe que para que la recta intercepte al plano, debe existir un punto $P(r)$ de esta que cumpla la ecuación del plano. Por ello, se define el sistema de ecuaciones conformado por la

ecuación del plano y la de la recta, $P(r) = P_0 + r(P_F - P_0)$, y se obtiene el parámetro r [29], como se muestra en la ecuación 2.37, que permita obtener Q.

$$r = \frac{-D - P_0 \cdot \vec{N}}{(P_F - P_0) \cdot \vec{N}} \tag{2.37}$$

Una vez obtenido el punto de intersección, como solo nos interesa la intersección de un determinado segmento de la recta con un área específica del plano, hay que verificar su pertenencia a ambos. En el caso del segmento, si el vector director de la recta utilizado engloba el segmento, bastaría con verificar $0 \leq r \leq 1$. Para la pertenencia al área, hay múltiples formas de probarlo, una de ellas es definir el área total A_T del triángulo definido por los 3 puntos dados del plano, y obtener las 3 áreas formados por el punto obtenido $P(r)$ y dos de los puntos dados. Si la suma de estas 3 áreas es igual al área total, el punto se encuentra en el área de interés.

$$A_T = A_1 + A_2 + A_3 \tag{2.38}$$

Verificada la pertenencia del punto, se pasaría a obtener el nuevo vector de la pelota como en los casos anteriores (2.5.1.1), siendo en este caso la normal del plano sobre el que la pelota incide la mostrada en la ecuación 2.34.

2.5.3 Choque esfera-esfera

En el siguiente apartado se analiza la colisión de 2 esferas en el espacio 3D.

Para verificar el choque entre dos esferas solo es necesario verificar el cumplimiento de una condición, siendo esta que la distancia entre los centros de ambas esferas debe ser inferior a la suma de sus radios,

$$D <= r_A + r_B \tag{2.39}$$

donde D representa el módulo de la distancia entre los centros de ambas esferas y las r representan los radios de las esferas.

Considerando que se toma de supuesto que solo la esfera A esta en movimiento, mientras que la B permanece inamovible, y que se estudia el movimiento durante un intervalo de tiempo t, es necesario verificar sus posiciones relativas la una de la otra múltiples veces en ese tiempo de la forma indicada en la ecuación 2.39 para asegurar que no se produzca un choque entre ellas.

En el caso de que en una de las verificaciones el choque de positivo, se procederá a determinar el nuevo vector director de la esfera A tras su choque con la esfera B, en la figura 2.17 se muestra un ejemplo simplificado a 2 dimensiones del efecto de este choque. Como en los casos anteriores, el nuevo vector director se obtiene mediante el método explicado en el apartado 2.5.1.1, tomando para este caso como normal, la normal al plano tangencial a ambas esferas en el momento del choque, que se corresponde con el vector definido por los centros de ambas esferas, como se muestra en la siguiente ecuación:

$$\vec{N}_{esferas} = \frac{P_A - P_B}{|\vec{N}_{esferas}|} \tag{2.40}$$

donde P_A y P_B representan los centros de las esferas, siendo la esfera A la que esta en movimiento.

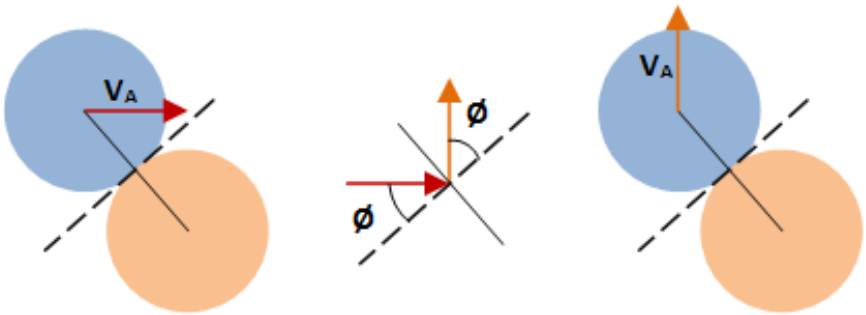


Figura 2.17: Representación 2D de la variación del vector director ante una colisión entre dos esferas.

Capítulo 3

Desarrollo

3.1 Descripción del algoritmo

En este TFG, como se mencionó en el capítulo de introducción (1), se genera una librería que permita la visualización de los usuarios y su interacción con elementos de un entorno virtual. El algoritmo desarrollado para ello se compone de 2 procesos: el principal y el secundario, conectados e independientes entre sí, como se puede ver en el esquema de la figura 3.1, presente en el capítulo 1 y que se muestra de nuevo para facilitar la lectura. El proceso principal es el encargado de regular el procesamiento de los datos de los usuarios que se reciban, a través de la identificación de su presencia y la detección de posibles interacciones de estos con el entorno. El proceso secundario se encarga de proporcionar un acceso a los datos de profundidad de la cámara, mediante la visualización de estos en una ventana secundaria. Cada uno de estos procesos cuenta con distintas etapas que les permiten cumplir con sus respectivas funcionalidades, las cuáles se describen en mayor detalle a continuación.

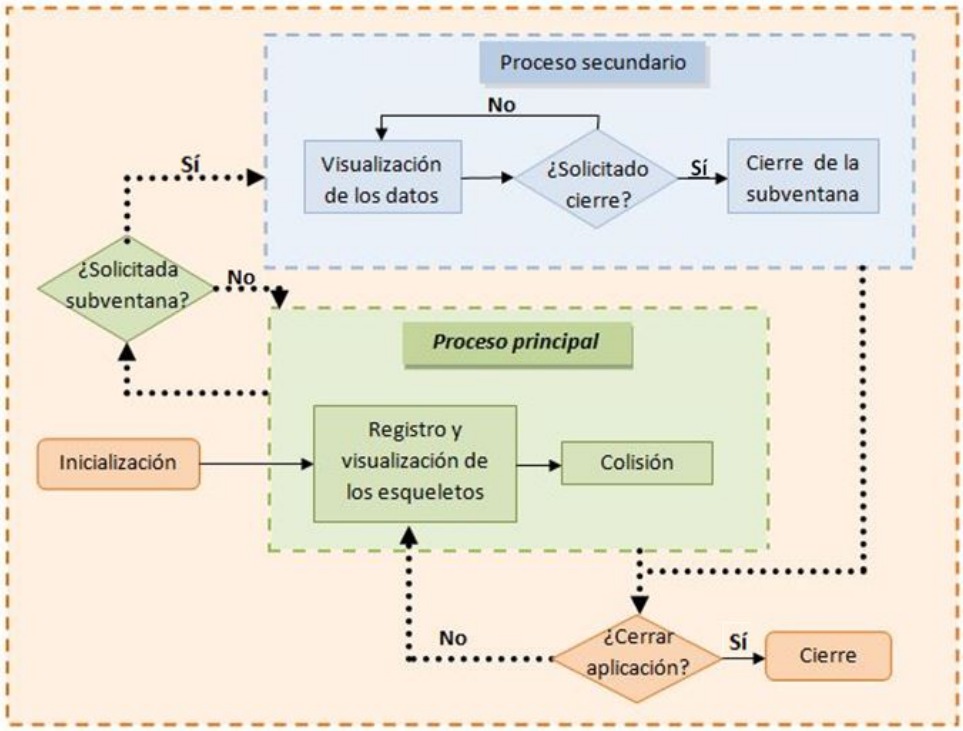


Figura 3.1: Diagrama de bloques general del algoritmo desarrollado.

- **Inicialización**. El sistema desarrollado, compuesto por los 2 procesos mencionados, requiere el uso de ciertos elementos externos, tanto de software como de hardware. Por ello en esta fase, se procederá a garantizar la correcta conectividad y calibración de estos elementos, de forma que los restantes procesos del algoritmo puedan acceder a ellos fluidamente y sin errores. Para ello, esta fase verifica que exista una cámara ToF, en este caso una kinect v2, conectada al equipo y se encarga de configurarla a las características deseadas para la ejecución del proyecto. Así mismo, también prepara y configura las librerías de apoyo del algoritmo desarrollado, en concreto OpenNI y NiTE.

- **Visualización de los datos**. En el proceso secundario del algoritmo desarrollado, se solicita la visualización de los datos que están siendo obtenidos por la kinect v2, solicitud que se cumplimenta mediante el desarrollo de esta fase. En ella, se procede al procesamiento de los datos de la cámara para que su visualización sea posible a través del entorno de desarrollo OpenGL, para lo cuál los datos proporcionados se deben texturizar. Como añadido, esta fase también permite alterar el tamaño de la ventana de visualización, a través de un evento asociado a ella.

- **Registro y visualización de los esqueletos**. El objetivo fundamental de este TFG es la visualización de los esqueletos de los usuarios, objetivo que se alcanza con la implementación de esta fase. En ella, mediante la lectura de los datos proporcionados por la kinect, se identifica a los usuarios presentes y su respectiva visibilidad. De cumplir con las condiciones necesarias para considerarse visibles, se procede a la obtención de sus esqueletos corporales y su posterior visualización en el entorno virtual. Esta fase conforma la primera parte del proceso principal mencionado con anterioridad.

- **Colisión**. Esta fase, que conforma la segunda y última parte del proceso principal, es la encargada de cubrir el segundo objetivo del TFG, siendo este la integración en el algoritmo desarrollado de la posibilidad de que los usuarios que sean detectados puedan interactuar con el entorno virtual. En ella, se toman los datos de los esqueletos obtenidos en la fase de *registro y visualización de los esqueletos* y se verifica, mediante el análisis de los casos de intersecciones geométricas mencionadas en 2.5, si se produce colisión entre los esqueletos y el objeto del entorno virtual designado, en concreto una pelota.

- **Cierre**. Al igual que en la fase de inicialización, esta fase afecta a ambos procesos. Del mismo modo que se requería inicializar los sistemas, también es necesario cerrarlos, mediante la liberación de todos los recursos configurados, tanto de memoria reservada para la aplicación, como de objetos que se hayan creado para tratar con los datos adquiridos por la cámara. En este caso, la etapa de cierre se ha dividido en dos fases para facilitar su manejo y agilizar la ejecución de la aplicación desarrollada, siendo estos el cierre de la ventana designada para la visualización de los datos y el cierre completo de la librería. El primero se encarga exclusivamente de la liberación de los recursos asociados al desarrollo del proceso secundario, mientras que el segundo bloque se encarga de la liberación de todos los recursos, incluidos los que se liberan con el primer bloque.

A continuación, se describen cada una de las fases aquí mencionadas en más detalle.

3.2 Inicialización del sistema

Como se ha descrito en el apartado anterior, la fase de inicialización es la encargada de configurar los recursos de software y de hardware necesarios para la ejecución de la librería desarrollada. Para ello se ha generado una función, llamada `_initialization()`, que se encarga de implementar esta funcionalidad.

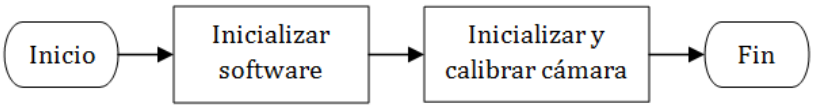


Figura 3.2: Diagrama de flujo de la función de inicialización del algoritmo desarrollado, `_initialization()`.

En esta fase, como se ve en la figura 3.2, se realiza la inicialización de las librerías, en específico de OpenNI y NiTE, para lo cual se realiza la llamada a las funciones de inicialización respectivas a cada librería y que, como se describía en el capítulo 2 (2.3), son necesarias para poder acceder al resto de los recursos y funcionalidades proporcionados por estas. Una vez verificada la correcta inicialización del software mencionado, se procede a la inicialización y calibración del dispositivo, en este caso una cámara ToF kinect v2, para lo cual se ha desarrollado un algoritmo cuyo diagrama de flujo puede verse en la figura 3.3 y que se corresponde con el código contenido en la función `initKinect()`. Como se puede ver en el diagrama mencionado, la inicialización de la kinect y su respectiva calibración sigue el siguiente proceso:

1. Conexión al dispositivo:
   Una vez que se ha verificado la inicialización correcta del software, se debe asegurar que exista un dispositivo conectado y en posesión de las características deseadas para el funcionamiento del código implementado. Para realizar esta verificación se hace uso de la librería OpenNI, en específico de sus funciones `openni::Device::open()` y `openni::Device::hasSensor()`. La función `open()` se encarga de abrir un canal de comunicación con el dispositivo que se le indique, y la función `hasSensor()` verifica que el dispositivo posea el tipo de sensor especificado, en este caso un sensor de profundidad.

2. Configuración del sensor:
   Tras conectarse al dispositivo, el siguiente paso es la configuración del sensor de profundidad detectado. Para ello, el primer paso es la asignación de este sensor al dispositivo, lo que se realiza a través de la función `openni::VideoStream::create()`. Una vez asociados se procede a la configuración del sensor para que actúe con las características deseadas, siendo estas: 30 fps, resolución de 640x480 y formato de pixel *DEPTH_1_MM*, características que se determinan mediante la llamada a las funciones `setFps()`, `setPixelFormat()` y `setResolution()` de la clase `openni::VideoMode`. Tras pasar estos datos como argumento a la función `openni::VideoStream::setVideoMode`, el único punto faltante para la finalización de la calibración del sensor es su activación, lo que iniciará la toma de datos de la cámara de forma que la aplicación tenga acceso a ellos. Esto se realiza a través de la llamada de la función `openni::VideoStream::start()`. Como se ha mencionado anteriormente, la resolución de profundidad de la kinect es de 512x424 y no de 640x480 como se fija en la calibración. El motivo de que la resolución de profundidad se fije a 640x480 es que la librería NiTE requiere esta resolución para poder funcionar.

En esta fase, también se realiza la asociación de un objeto `nite::UserTracker` al dispositivo en uso, para su posterior utilización en la obtención de los datos que permitan identificar a los usuarios; así como, se verifica que los ejes del sistema de referencia de la kinect proporcionados sean unitarios.

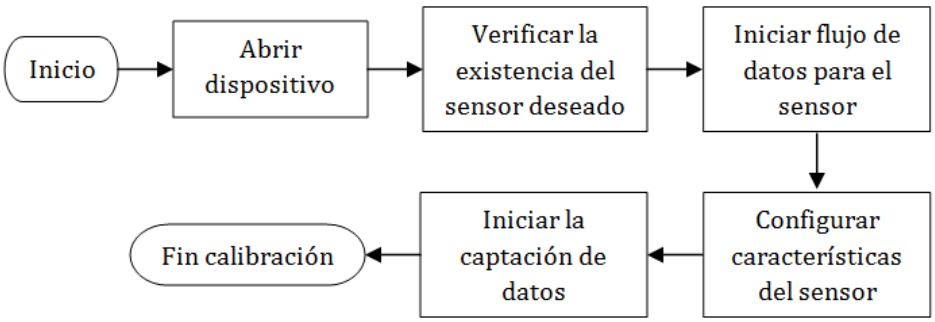


Figura 3.3: Diagrama de flujo de la función de inicialización y calibración de la cámara, `initKinect()`.

3.3 Visualización de los datos por la ventana secundaria

Esta fase, junto a la descrita en el apartado 3.6.1, componen el proceso secundario del algoritmo desarrollado. Este proceso es el encargado de crear una ventana que permita la visualización de los datos que la cámara esta obteniendo y de manejarla hasta que se solicite su cierre, ya sea el cierre específico de la ventana o el cierre de la aplicación.

Para el cumplimiento de la funcionalidad deseada se han creado múltiples funciones, siendo estas: `_display()`, `processKeys()`, `processMouseEvent()`, `renderSceneSubWin()` y `_changeSizeSubWin()`. El objetivo y la funcionalidad específicas de cada una de ellas se explica a continuación.

Lo primero que se ha de hacer para poder visualizar los datos es crear la ventana, de lo cual se encarga la función `_display()`. Esta función verifica que el acceso a los datos del dispositivo sea correcto, y determina los parámetros de la ventana a crear, siendo estos el tamaño y la posición de esta, dados en relación a la ventana principal. Respecto al tamaño, en este caso se ha optado por fijar únicamente el ancho de la ventana, de tal forma que su tamaño sea relativo al de la ventana principal y, a su vez, que la altura se determine en función del ancho, de tal forma que la ventana mantenga siempre su dimensionalidad, siendo el parámetro que determina esta dimensionalidad W_H_RATIO, que en este caso se ha tomado como

$$W\_H\_RATIO = \frac{Altura\ imagen\ profundidad}{Anchura\ imagen\ profundidad} = \frac{424}{512} \tag{3.1}$$

De esta forma, la ventana que se obtiene inicialmente se corresponde con la mostrada en la figura 3.4, donde $W_{datos} = W_{principal} * widthRatio$ con $widthRatio = 0{,}4$, $H_{datos} = W_{datos} * W\_H\_RATIO$, $P_X = W_{principal} * xPosPercent$ con $xPosPercent = 0{,}58$ y $P_Y = H_{principal} * Y\_POS\_SUB\_PERCENT$ con $Y\_POS\_SUB\_PERCENT = 0{,}04$. En el algoritmo desarrollado las anchuras y las alturas de las ventanas se declaran mediante las variables: mainWinW, mainWinH, subWinW y subWinH.

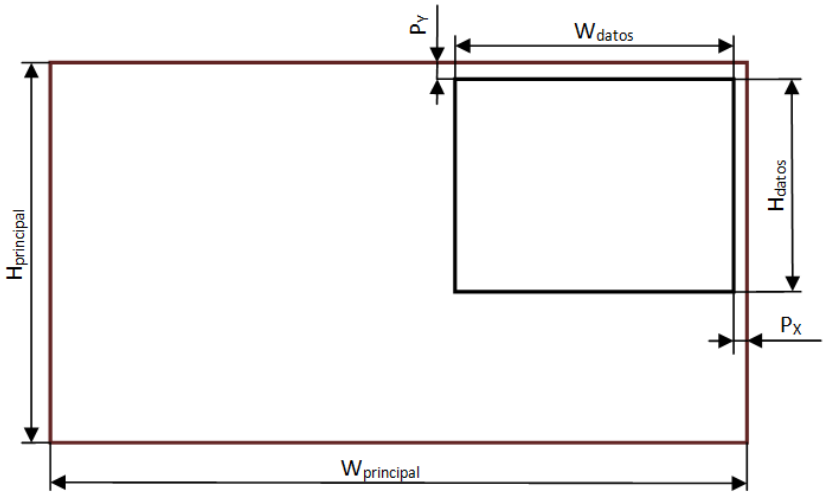


Figura 3.4: Tamaño y posición de la ventana de visualización de datos respecto a la principal.

Una vez generada la ventana, se procede a la reserva de la memoria necesaria para almacenar los datos de la textura y los datos de las articulaciones de los esqueletos.

Además de lo mencionado toda ventana tiene una serie de eventos que se le pueden asociar, en este caso, los eventos asociados son el de teclado, el de ratón y el de visualización, cuyas funciones asociadas son `processKeys()`, `processMouseEvent()` y `renderSceneSubWin()`, respectivamente. El funcionamiento de cada una de estas funciones se explica en más detalle a continuación.

La función pasada como argumento a la función `glutKeyboardFunc()` determina a qué función llamará el código cuando se produzca un evento de teclado, estando sobre la ventana a la cual el evento este asociado. En este caso, como se ha mencionado, la función designada ha sido `processKeys()`. Esta función únicamente responde ante la pulsación de la tecla *ESC*, ante lo cual iniciará el cierre de la aplicación, misma acción que ocurrirá si se pulsa la misma tecla sobre la ventana principal.

Al igual que con el teclado, también se puede asociar una función a una acción del ratón mediante su paso como argumento a la función `glutMouseFunc()`. En este caso la función creada ha sido `processMouseEvent()`, que responde ante la pulsación de los botones del ratón modificando el tamaño de la ventana de visualización, de la forma siguiente:

- Botón derecho
  Ante una pulsación de este botón, la función verifica el tamaño actual de la ventana para determinar que no esta en su tamaño mínimo permitido y, de ser así, reduce su tamaño.

- Botón izquierdo
  Este caso aplica el caso contrario, la función verifica que la ventana no este en su tamaño máximo y, si es así, aumenta el tamaño de la ventana.

El incremento y decremento mencionados se han fijado a un 5 %, es decir, que si la ventana de visualización esta calibrada para ser equivalente al 40 % del tamaño de la ventana principal, al realizar una pulsación sobre el ratón su tamaño se verá modifica al 35 % o al 45 %, en función del botón pulsado. Así mismo, el máximo y el mínimo mencionados se corresponden con los valores designados por las constantes $SUB\_WIN\_W\_MAX$ y $SUB\_WIN\_W\_MIN$, que se han declarado como 512 y 320, respectivamente.

La función encargada de realizar el cambio de tamaño mencionado es `_changeSizeSubWin()`. Además de la forma mencionada como activación de esta función (el evento de ratón de la ventana de datos), ya que la ventana de visualización es dependiente de la principal, esta función también se activa de producirse un cambio en el tamaño de la ventana principal, cuya función de cambio de tamaño es `changeSize` que esta asociada a la función `glutReshapeFunc` presente en el código desarrollado en `ispaceVirtual.c`. La función `_changeSizeSubWin()`, como se ha mencionado, se encarga de calcular y actualizar los parámetros de tamaño y posición de la ventana de visualización y verifica si el tamaño resultante está permitido o no. Si la ventana es visible y el tamaño es menor al permitido ocultará la ventana, y si la ventana está oculta y el tamaño resultante es superior al límite establecido la mostrará.

Finalmente, la función `renderSceneSubWin()` asociada a la función `glutDisplayFunc()` de la ventana de datos, es la encargada de determinar que se mostrará en la ventana creada y su llamada se realiza desde la función `renderSceneAll` del código `ispaceVirtual.c`, siempre y cuando la ventana de datos este activa. En esta función, se realiza la lectura de los datos de profundidad, datos que se almacenan en una variable designada para el almacenamiento de la textura para su posterior visualización por pantalla. Ya que el tipo de dato de los datos de profundidad (DepthPixel = 2 bytes) y el necesario para la visualización por pantalla (RGB888Pixel = 3 bytes) son distintos, esta textura debe generarse de una forma específica mostrada en el algoritmo 3.1. Además de mostrar los datos de profundidad, esta función se encarga de mostrar el esqueleto de los usuarios presentes una vez que han sido identificados por la función `skeltrack()` descrita en el apartado 3.4, para lo cual convierte las posiciones de las articulaciones determinadas por la kinect en un espacio 3D en posiciones 2D respectivas a la imagen de profundidad.

Data: Datos de la imagen de profundidad (640x480) proporcionados por la cámara ToF.
Result: Variable *texture* que contiene la textura creada a partir de los datos de la cámara.
Creación de un histograma *depthHist* de variables tipo *RGB888Pixel* que refleje la frecuencia de aparición de cada medida de profundidad leída, para una resolución de 512x424;
Declaración de variables: anchuraImagen = 512, alturaImagen = 424, pData = puntero al conjunto de datos proporcionados por la imagen (640x480), filaTex=512 y filaData = 640;
for *y = 0 hasta y = alturaImagen* **do**
    texturePixel = Puntero a la zona de memoria a escribir, siendo su valor = texture + y*filaTex;
    for *de x=0 hasta x=anchura de la imagen, incrementar texturePixel* **do**
        depthPixel = Puntero a la zona de memoria que contiene los datos, siendo su valor = pData + y*filaData + x;
        if *depthPixel* $\neq 0$ **then**
            depthValue = (depthHist[*depthPixel] / número de lecturas de profundidad válidas)*255;
            texturePixel = tonalidad gris, valor = 255 - depthValue;
        else
            texturePixel = se define de color negro [RGB=(0,0,0)];

Algoritmo 3.1: Algoritmo desarrollado para la texturización de la imagen de profundidad proporcionada por la kinect.

3.4 Registro y visualización de los esqueletos por la ventana principal

Esta fase, que representa el primer bloque de funciones del proceso principal del algoritmo desarrollado, es la encargada de cubrir el objetivo fundamental de este proyecto, siendo este la visualización de los esque-

letos de los usuarios sobre un entorno virtual. La función principal asignada a esta fase es `skeltrack()`, en el diagrama siguiente se muestran los procesos en los que se divide esta función, los cuales se explican con más detalle a continuación:

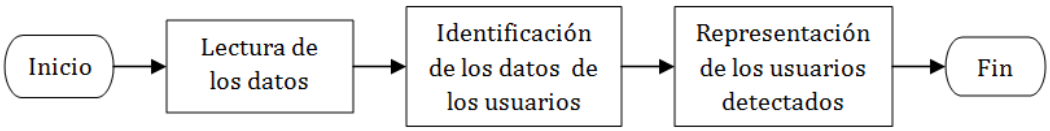


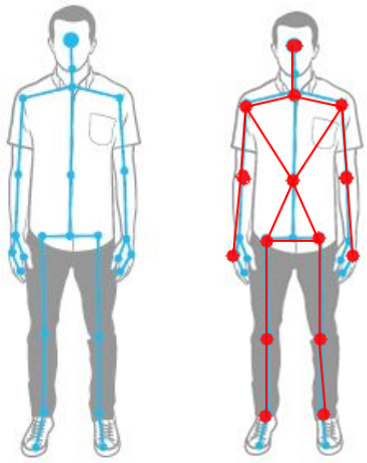
Figura 3.5: Diagrama de bloques de la función `skeltrack()`

Como se puede ver en el diagrama anterior, esta fase se puede sintetizar en 3 procesos distintos:

1. Lectura de los datos de profundidad de la cámara.
2. Identificación de los usuarios presentes y obtención de los datos asociados a ellos.
3. Representación de los usuarios presentes.

Los dos primeros procesos representan la obtención de una de las frames obtenidas por la cámara, entendiendo frame como el conjunto de los datos que componen una imagen de profundidad obtenida por la kinect, para la posterior identificación entre estos datos de aquellos conjuntos que hayan sido designados como personas, lo que permite la posterior obtención de las articulaciones que conforman sus respectivos esqueletos.

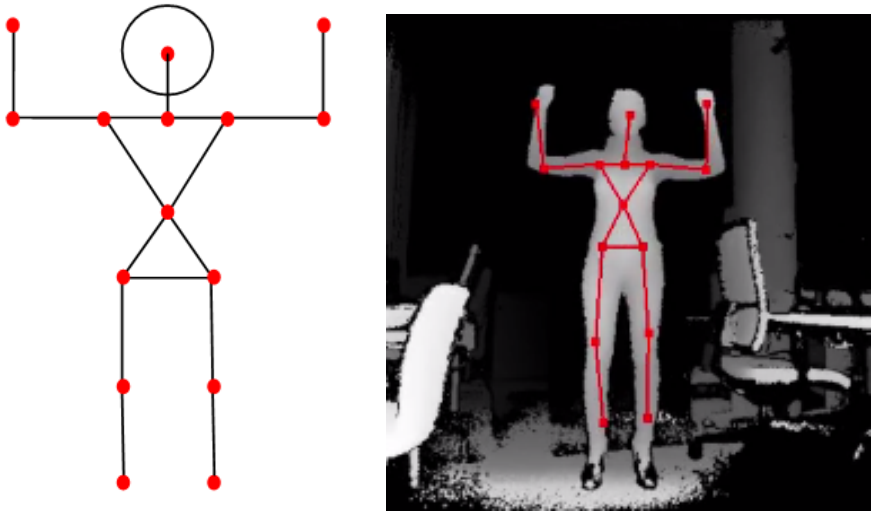
Una vez obtenidos estos datos, y solo en el caso de que se haya identificado a alguna persona en la escena, se activará el tercer proceso, por el cual se visualizara a los usuarios presentes mediante la representación en el entorno virtual de sus esqueletos, esqueletos que quedan determinados por la información de las articulaciones obtenida de los conjuntos de datos proporcionados por la ejecución del segundo proceso. Para este proyecto y aunque la kinect v2 puede detectar hasta 25 articulaciones, debido a limitaciones impuestas por el software utilizado, se integran únicamente 15 articulaciones, que se sintetizan de la forma mostrada en la figura siguiente.



(a) Captado por la Kinect v2. (b) Utilizado en el algoritmo.

Figura 3.6: Representación del esqueleto.

La forma de identificar estas articulaciones y poder realizar un seguimiento activo de su posición en el tiempo es a través del proceso de calibración. Este proceso se puede activar mediante 2 métodos: la autocalibración o la identificación por pose. La autocalibración permite la identificación de la persona y la correspondiente calibración de su esqueleto desde el momento en que esta persona entre en el campo visual de la cámara y siempre que su postura permita una clara identificación de ella. Por el otro lado, la identificación por pose exige que el usuario presente se posicione en una postura determinada para poder iniciar la calibración de su esqueleto, en este caso concreto se hace uso de la pose PSI, mostrada en la figura 3.7.



(a) PSI (b) Ejemplo práctico.

Figura 3.7: Pose PSI

En el algoritmo desarrollado, se da opción a utilizar ambos métodos y a modificar el que se esta usando en cualquier momento de la ejecución del programa, siendo el método por defecto la autocalibración. La forma de modificar el método que se está utilizando se describe en el *Manual de usuario* (D.2). En base al método utilizado el comportamiento del algoritmo se modifica ligeramente de la forma siguiente:

- Si se opta por la autocalibración, por cada usuario detectado se realizarán los siguientes pasos: primero se verificará si el usuario aparece por primera vez en la escena y de ser así se iniciará la captura y seguimiento del esqueleto; después, se verificará si el usuario detectado ya no esta en la escena y de ser así se detendrá su seguimiento; y, finalmente, se verificará que el usuario sea visible, es decir, que la kinect sea capaz de detectarlo correctamente, si es así se solicitarán los datos de las articulaciones.
- Si se opta por la identificación por pose, en cambio: primero se verificará si el usuario aparece por primera vez en la escena y de ser así se iniciará la detección de la pose designada (PSI); después, se verificará si el usuario detectado ya no esta en la escena y de ser así se detendrá la búsqueda que este activa (detección de la pose o seguimiento del esqueleto); finalmente, se verificará que el usuario sea visible, de ser así, se comprobará si el usuario acaba de posicionarse en la pose designada, caso en el que se inicia la captura y seguimiento del esqueleto; así como, se solicitarán los datos de las articulaciones.

A continuación se muestra el diagrama de funcionamiento de la función `skeltrack()` (figura 3.8), donde se muestra explícitamente los pasos seguidos para ambos casos, para facilitar la comprensión de la diferencia existente entre ambos.

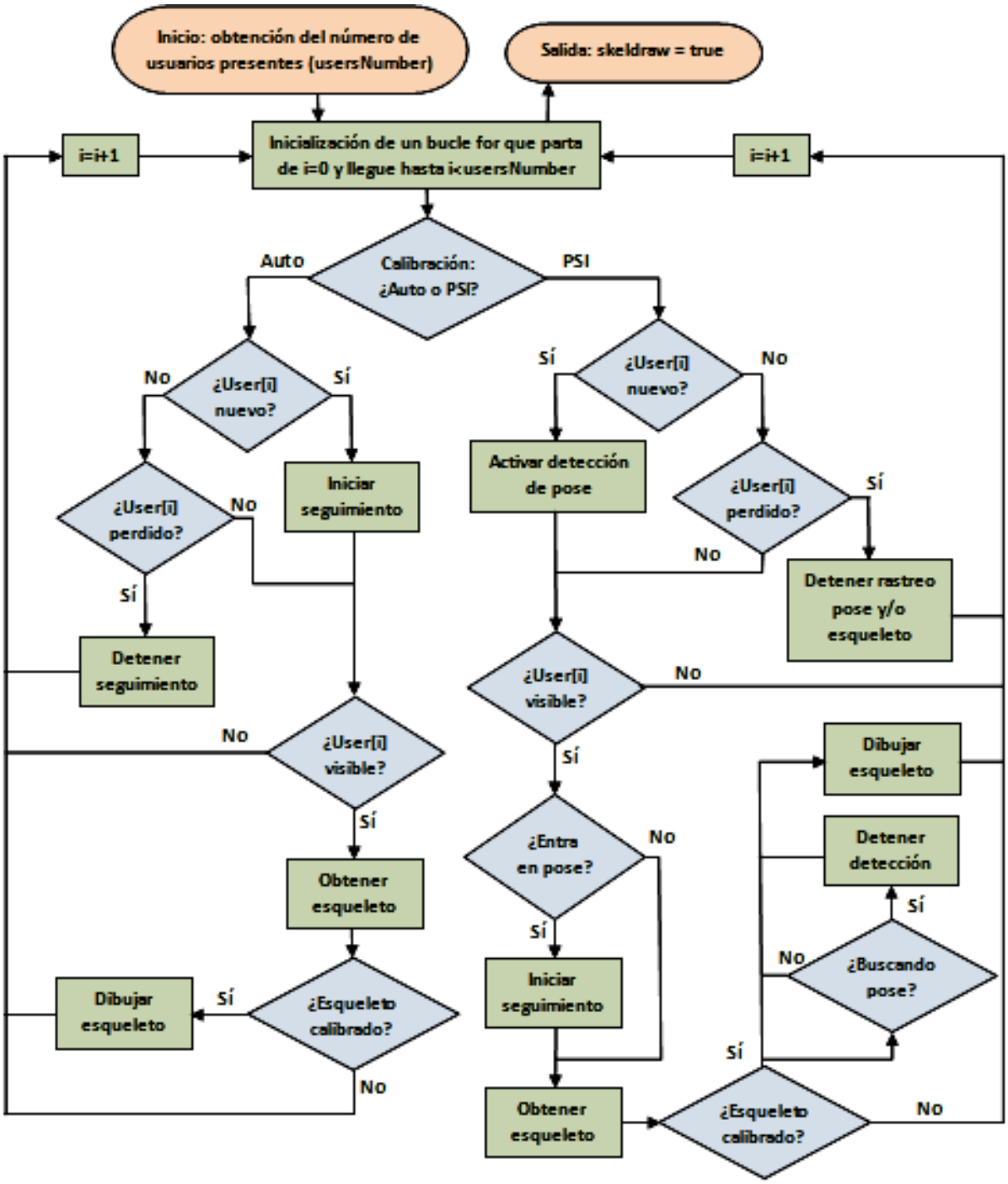


Figura 3.8: Diagrama de funcionamiento de la función `skeltrack()`: métodos de calibración.

Una vez identificadas las articulaciones que se van a utilizar y antes de iniciar el proceso de dibujo de estas, se debe tener en cuenta que toda la información relativa a su posicionamiento y orientación proporcionada por la kinect, viene referenciada a esta y no al entorno virtual en el que se desean representar los esqueletos. Es por ello, que se debe realizar una transformación del sistema de referencia (SR) de estos datos antes de su representación. En este caso concreto, se ha optado por utilizar las MTH, que como se explicaba en el capítulo 2 (2.4) representan uno de los métodos de representación espacial existentes y permiten representar la transformación de un vector entre 2 sistemas de coordenadas. Una MTH se construye de la forma mostrada en la ecuación 2.10 presente en el capítulo previamente citado, pero que se muestra aquí de igual forma para facilitar la lectura.

$$T = \begin{bmatrix} s_x & n_x & a_x & r_x \\ s_y & n_y & a_y & r_y \\ s_z & n_z & a_z & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

donde $\vec{s}$, $\vec{n}$ y $\vec{a}$ representan las proyecciones de los ejes del sistema de la kinect sobre el sistema del entorno y componen a su vez la matriz de rotación, y $\vec{r}$ representa el vector de traslación.

Aplicando este método, se pueden obtener las coordenadas de cada articulación sobre el sistema de referencia del entorno. En el algoritmo desarrollado, sin embargo, se ha optado por no generar la matriz como tal para facilitar el manejo, ya que los datos que proporciona la kinect no son homogéneos y por lo tanto habría que convertir las estructuras en las que se almacenan a vectores homogéneos para poder operar con ellos. Por ello, se ha optado por una aplicación directa de la ecuación resultante al aplicar la transformación, ecuación que deberá aplicarse a cada articulación identificada. Expresándolo de una forma genérica, la ecuación mencionada queda de la forma siguiente:

$$\begin{aligned} x_e &= s_x x_k + n_x y_k + a_x z_k + r_x \\ y_e &= s_y x_k + n_y y_k + a_y z_k + r_y \\ z_e &= s_z x_k + n_z y_k + a_z z_k + r_z \end{aligned} \tag{3.3}$$

donde, como se ha mencionado previamente, $\vec{s}$ representa la proyección del eje X del SR de la kinect sobre el del entorno, al igual que $\vec{n}$ representa la del eje Y y $\vec{a}$ la del eje Z, y $\vec{r}$ es el vector de translación entre sus orígenes. Estos vectores quedan representados en el algoritmo desarrollado mediante las estructuras **x_kinect**, **y_kinect**, **z_kinect** y **kinectPos**, respectivamente.

Finalmente, una vez obtenidos los datos de las articulaciones, se ha implementado un bucle que permite la obtención de las tasas de confianza de las medidas obtenidas, tanto de orientación como de posición, mediante la implementación de las funciones `getPositionConfidence()` y `getOrientationConfidence()` de la librería NiTE, que se mostrarán por pantalla para su posterior estudio como parte de los resultados obtenidos (capítulo 4).

3.5 Identificación y manejo de las colisiones

En esta fase se integran los procedimientos necesarios para la cumplimentación de la parte faltante del objetivo para el cual se ha desarrollado este TFG. El objetivo mencionado se trata de la implementación en el sistema desarrollado de la posibilidad de que los usuarios sean capaces de interactuar con elementos del entorno virtual, a través de su propio movimiento.

En concreto el código desarrollado, implementado mediante la función `checkCollisionSkel()`, permite la interacción de los usuarios detectados con una pelota en movimiento dentro del entorno virtual. De tal forma que, en el caso de que ambos objetos (la pelota y el esqueleto) colisionen, se produzca una reacción acorde que modifique la ruta de movimiento seguida por la pelota. Para ello se han considerado 3 casos de colisión en el espacio, como se describe en el capítulo 2 (2.5), siendo estos:

- La colisión entre la cabeza del usuario y la pelota.
- La colisión de la pelota con las líneas representativas de los huesos del esqueleto.
- La colisión de la pelota con el área que conforma el torso del esqueleto.

Cada uno de estos casos se implementa en un bucle propio que verifica si esa colisión se ha producido para cada esqueleto representado, de tal forma, que el algoritmo creado se compone de 3 bucles independientes que parten de los datos obtenidos en la fase de *Registro y visualización de los esqueletos* (3.4), como se puede ver en el diagrama de la figura 3.9 mostrado a continuación.

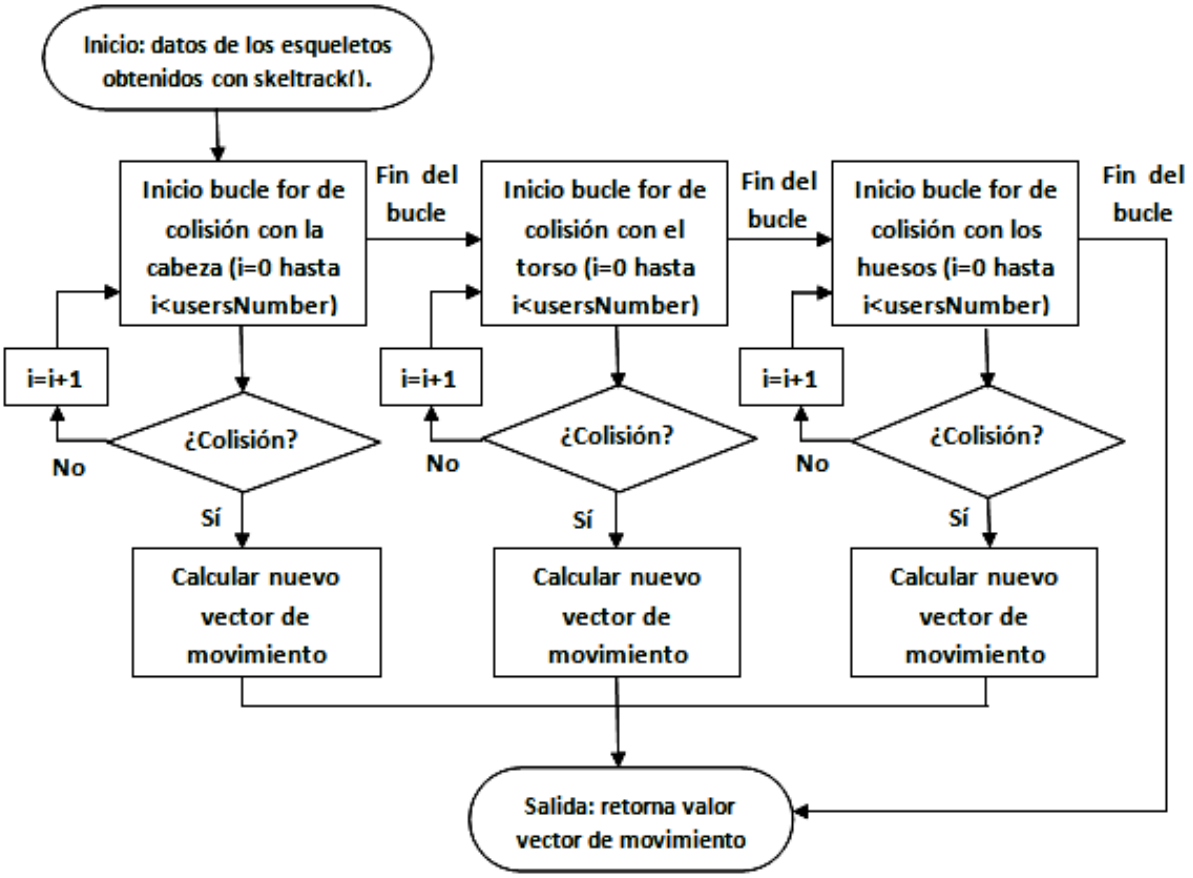


Figura 3.9: Diagrama de flujo de la función `checkCollisionSkel()`.

El algoritmo desarrollado para cada uno de estos bucles, cuya estructura general se muestra en el algoritmo 3.5 presente al final de este apartado y que representa la estructura base desde la que se parte para el desarrollo de cada uno de los casos específicos, se puede resumir en 3 grandes procesos:

1. **Detección de la colisión**
   Con este proceso se determina si las posiciones de ambos cuerpos coinciden en el espacio, las condiciones que verifican este hecho cambian en función del caso de colisión que se este analizando, de esta forma:

   - Si se esta analizando la colisión con la cabeza se parte de la condición de que la distancia entre sus centros debe ser inferior a la suma de sus radios. Para ello, se divide el vector de movimiento en 10 partes y se comparan esas 10 posiciones de la pelota con el centro de la

cabeza, lo que se implementa mediante un bucle do-while como el siguiente:

Data: Posición de la cabeza del esqueleto $P_{cabeza} = users.heads[i]$ y datos de la pelota (posición $P_{pelota} = ball \rightarrow coord$ y vector de movimiento $\vec{V_T} = ball \rightarrow speed$).
Result: Determinación de si se produce colisión con la cabeza.
distancia = radio cabeza + radio pelota;
repeat
  bc = distancia entre P_{cabeza} y P_{pelota};
  if $|bc| <= distancia$ **then**
    Se indica que se produce colisión y se sale del bucle do-while;
  Se actualiza la posición de la pelota sumándole ($0{,}1 * V_T$);
  Se define el vector desde el punto original de la pelota hasta el actual ($\vec{v}$);
until $|\vec{v}| <= |\vec{V_T}|$
;

Algoritmo 3.2: Bucle do-while para la detección de la colisión con la cabeza.

- Si se esta evaluando la colisión con el torso, se parte del plano definido por las articulaciones de los hombros y la del torso (plano superior) y por el definido por las articulaciones del torso y la cadera(plano inferior), cuyos parámetros (normal $\hat{N}$ y constante D) se calculan mediante las fórmulas

$$Nplano = \frac{PvecPlano}{|PvecPlano|} \; con \; PvecPlano = vTAi \times vTAd \tag{3.4}$$

$$D = -(Nplano \cdot users\_body[i].torso) \tag{3.5}$$

donde $vTAi$ representa el vector entre la articulación izquierda correspondiente (hombro o cadera) y el torso y $vTAd$ representa lo mismo que la anterior pero para la articulación de la derecha. Para cada uno de estos planos se verifica que la pelota no se mueve en una dirección paralela a ellos, mediante la aplicación del producto escalar entre la normal al plano ($NplanoS$ o $NplanoI$) y el vector de movimiento de la pelota ($ball \rightarrow speed$). El resultado de esta operación debe ser distinto de 0, ya que si diera 0 implicaría que $vector \perp Normal$, es decir, que el vector estaría contenido o sería paralelo al plano definido.

$$PescPP = (ball \rightarrow speed) \cdot Nplano \neq 0 \tag{3.6}$$

Si se cumple la condición anterior, se procede a calcular el punto de colisión con el plano respectivo, para lo cual se define el parámetro r, que determina el punto de colisión $ColPt$ al aplicarlo sobre la ecuación de la recta de la pelota.

$$r = \frac{-(D + (ball \rightarrow coord) \cdot Nplano)}{(ball \rightarrow speed) \cdot Nplano} \tag{3.7}$$

$$ColPt = (ball \rightarrow coord) + r \cdot (ball \rightarrow speed) \tag{3.8}$$

Por último, se verifica si el punto obtenido pertenece al área definida por el torso, lo que en este caso se realiza mediante la comparación entre el área contenida por las 3 articulaciones que definen el plano (A_T) y el área resultante de la suma de las áreas que forman el punto de colisión con dos de las articulaciones mencionadas antes

$$A_{suma} = A_{Col-Art1-Art2} + A_{Col-Art2-Art3} + A_{Col-Art3-Art1} \tag{3.9}$$

si el punto está contenido en el plano ambas áreas serán equivalentes.

$$A_T = A_{suma} \tag{3.10}$$

Data: Posiciones de las articulaciones del torso (hombros y torso) y datos de la pelota (posición $P_{pelota} = ball \rightarrow coord$ y vector de movimiento $\vec{V}_T = ball \rightarrow speed$).
Result: Determinación de si se produce colisión con el torso.
Cálculo de los parámetros del plano superior ($\vec{N}_{TS}$ y constante D_S);
if $\vec{V}_T$ *no es paralelo ni coincidente al plano* **then**
  Obtención del punto de colisión plano-vector;
  if *Punto* $\in$ *torso superior y Punto* $\in$ *segmento* **then**
    Iniciar cálculo del nuevo valor de $\vec{V}_T$ (proceso 3) ;
    return $\vec{V}_T$;
Cálculo de los parámetros del plano inferior ($\vec{N}_{TI}$ y constante D_I);
if $\vec{V}_T$ *no es paralelo ni coincidente al plano* **then**
  Obtención del punto de colisión plano-vector;
  if *Punto* $\in$ *torso inferior y Punto* $\in$ *segmento* **then**
    Iniciar cálculo del nuevo valor de $\vec{V}_T$ (Proceso 3);
    return $\vec{V}_T$;

Algoritmo 3.3: Determinación de la colisión de la pelota con el torso del esqueleto.

- En el caso de la colisión con las líneas del esqueleto, huesos de ahora en adelante, se debe considerar que se dan 2 casos: que se intercepten o que se crucen. Para determinar cuál de los casos se esta dando, se debe verificar la coplanaridad entre los vectores directores de ambos segmentos, lo que se realiza mediante el cálculo del producto mixto. Para el cálculo de este producto se declaran los 3 vectores siguientes: vector director de la pelota ($ball \rightarrow speed$), vector director del hueso $bone = boneA - boneB$ y un vector que une ambos segmentos $Uph = boneA - (ball \rightarrow coord)$. Calculando el producto mixto entre ellos de la forma:

$$Pmixto = Uph \cdot [(ball \rightarrow speed) \times bone] \tag{3.11}$$

donde Uph y el vector resultante del producto vectorial (definido como $PvecHP$) se hacen unitarios antes de operar, si el resultado es 0 los vectores serán coplanarios y, tras descartar la posibilidad del paralelismo ($(ball \rightarrow speed) \times bone \neq 0$), se define que se esta en un caso de intersección. Si el resultado fuese distinto de 0, sería un caso de skew lines. Una vez definido el caso a tener en cuenta, se procede al cálculo de los parámetros que definen los puntos de interés, siendo estos el punto donde se produce la intersección ($ColPt$) o los puntos de la pelota y el hueso (PtP y PtH) en que los segmentos son más cercanos entre sí . Para la obtención de estos parámetros, definidos como k para el hueso y l para la pelota, se aplican las siguientes ecuaciones:

$$k = \frac{PvecUH \cdot PvecHP}{den} \tag{3.12}$$

$$l = \frac{PvecUP \cdot PvecHP}{den} \tag{3.13}$$

siendo $den = PvecHP.x^2 + PvecHP.y^2 + PvecHP.z^2$, $PvecUH$ el producto vectorial entre el vector Uhp y el vector director del hueso, $PvecUP$ el producto vectorial entre el vector Uhp y el vector director de la pelota, y $PvecHP$ el producto vectorial entre los vectores directores

de la pelota y el hueso. La definición de estos parámetros determina el final de la parte común a ambos casos (intersección y cruce), siendo los pasos siguientes determinados por el caso concreto que se este dando y que ha sido identificado con anterioridad, de esta forma:

- Si las líneas se intersecan, se calcula el punto de colisión $ColPt$ sustituyendo el valor de uno de los parámetros calculados en la ecuación correspondiente, en este caso:

$$ColPt = boneB + k \cdot bone \tag{3.14}$$

  Obtenido el punto, se procede a verificar que pertenezca a ambos segmentos, para lo cuál tanto k como l deben estar comprendidos entre 0 y 1. Ya que el volumen de la pelota no se esta considerando al realizar los cálculos, si el resultado de la verificación fuese negativo, se verificarían las siguientes condiciones:

  * La distancia entre el punto final del recorrido de la pelota y el punto de colisión calculado es inferior al radio de la pelota.

  * La distancia máxima entre los extremos del hueso y el segmento de la pelota es inferior al radio de la pelota

  que de cumplirse indicarían que sí se produce la colisión.

- Si las líneas se cruzan (skew lines) se verifica que el valor de los parámetros calculados se encuentre entre 0 y 1. De ser así, se procede a calcular los puntos PtP y PtH, que representan los puntos de los segmentos entre los que existe menor distancia. Por último, se verifica que esta distancia

$$disHP = |PtP - PtH| \tag{3.15}$$

  sea inferior al radio de la pelota, condición que confirma la colisión de la pelota con el hueso. Verificada la condición, se procede a determinar el punto del recorrido de la pelota que se encuentra a una distancia ($ball \rightarrow radius$) del punto del hueso PtH, y si este punto forma parte del segmento recorrido por la pelota.

En el algoritmo 3.4 mostrado a continuación, se han sintetizado los pasos explicados para facilitar su comprensión.

Data: Posiciones de las articulaciones del esqueleto y datos de la pelota (posición $P_{pelota} = ball \rightarrow coord$ y vector de movimiento $\vec{V}_T = ball \rightarrow speed$).
Result: Determinación de si se produce colisión con los huesos.

Declaración de los vectores Uph y $bone$;
Calculo del producto mixto (coplanaridad);
if $Pmixto == 0$ **then**
    Se verifica que las rectas no sean paralelas ($PvecHP$).;
    if $PvecHP == 0$ **then**
        Rectas paralelas, no se produce colisión;
        continue;
    else
        Las rectas se intersecan (intersection = true);
else
    Las rectas se cruzan (skew = true);

Cálculo de los parámetros k y l de las rectas;
if $intersection == true$ **then**
    Calcular punto de colisión $ColPt$;
    Verificar pertenencia del punto a los segmentos (incluyendo excepciones dadas);
    if $ColPt \notin segmentos$ **then**
        continue;
    Iniciar cálculo de la normal y el nuevo valor de $\vec{V}_T$ (Procesos 2 y 3);
if $skew == true$ **then**
    Verificar que el valor de los parámetros esta en el rango correcto;
    if $k \notin [0,1]$ o $l \notin [0,1]$ **then**
        continue;
    Calcular puntos PtP y PtH;
    Verificar que la distancia entre ellos ($DisHP$) no supere el valor del radio de la pelota;
    if $DisHP > ball \rightarrow radius$ **then**
        continue;
    Calcular Pd, punto del recorrido de la pelota a una distancia ($ball \rightarrow radius$) de PtH;
    Verificar si Pd es un punto del recorrido de la pelota (segmento);
    if $Pd \notin segmento$ **then**
        continue;
    Iniciar cálculo de la normal y el nuevo valor de $\vec{V}_T$ (Procesos 2 y 3);

Algoritmo 3.4: Determinación de la colisión de la pelota con los huesos.

2. **Cálculo de la normal al plano de colisión**
   Una vez determinado que se produce colisión y que esa colisión es válida ya que esta contenida en el área de interés, es decir, que el punto obtenido es parte del segmento definido como hueso o del área definida como torso o, en el caso de la cabeza, que los centros están a la distancia correcta; se procede al cálculo de la normal al plano de colisión, $\vec{N}$, que se tomará siempre como unitaria,

$\hat{N}$. Para ello, se debe determinar cuál es el plano de colisión. De esta forma, las normales que se obtienen se corresponden con:

- Colisión con la cabeza: se toma el plano de colisión como el plano tangencial a ambas esferas en el momento en el que se ha determinado que se produce colisión. Por ello, en este caso, la normal se corresponde con el vector de unión de los centros de ambas esferas en el momento de la colisión, vector definido en el proceso anterior y que se corresponde con $\vec{bc} = newpos - users.heads[i]$, donde $newpos$ representa la posición del centro de la pelota cuando se determina la colisión y $users.heads[i]$ es la posición de la articulación de la cabeza del usuario i. De esta forma se obtiene:

$$\hat{N}_{esferas} = \hat{bc} = \frac{\vec{bc}}{|\vec{bc}|} \tag{3.16}$$

- Colisión con el torso: el torso, como se ha definido con anterioridad, se define mediante 2 planos, uno para la parte superior (definido por las articulaciones de los hombros y el torso) y otro para la parte inferior (definido mediante las articulaciones de la cadera y el torso), por ello, los cálculos a continuación descritos se llevan a cabo 2 veces, una para cada plano. Como se ha mencionado, lo primero es determinar el plano de colisión que, en este caso, se corresponde con el plano que define el torso. Ya que un mismo plano se puede definir mediante 2 normales, cuyos sentidos son opuestos entre sí, para poder definir la normal correcta, es decir, aquella cuyo sentido este orientado hacia la pelota, hay que determinar a qué lado del plano se haya esta. A continuación, se describen los cálculos realizados para el caso del torso superior:

  (a) Cálculo de la normal (realizado durante el primer proceso):

  - Definición de los vectores del plano.

$$\begin{cases} vTHd = P_{hombrod} - P_{torso} \\ vTHi = P_{hombroi} - P_{torso} \end{cases} \tag{3.17}$$

  - Obtención de la normal: producto vectorial de vTHd y vTHi.

$$NplanoS = \frac{PvecPS}{|PvecPS|} \; siendo \; PvecPS = vTHi \times vTHd \tag{3.18}$$

  (b) Definición del sentido de la normal:

  - Definición del vector de aproximación (plano→pelota), vApro.

$$vApro = (ball \rightarrow coord) - ColPt \tag{3.19}$$

    siendo $ColPt$ el punto de colisión obtenido.
  - Verificación del sentido de la normal: se realiza el producto escalar entre la normal definida y el vector de aproximación, si el resultado es negativo, se invierte el sentido de la normal, multiplicando $NplanoPS$ por -1.

$$PescNvA = NplanoPS \cdot vApro \tag{3.20}$$

- Colisión con los huesos: como se ha descrito con anterioridad, en este tipo de colisión se consideran 2 casos: la intersección y las skew lines. En el caso de la intersección, la normal del plano de colisión queda definido como el producto vectorial del hueso y la normal del plano

que contiene los vectores del hueso y la pelota. Mientras que en el caso de las skew lines, la normal se define como la recta que es perpendicular al hueso y contiene el punto del vector director de la pelota que se encuentra a una distancia $D = R_{pelota}$. De esta forma, las normales obtenidas en ambos casos son:

(a) Normal para el caso de la intersección:

- Calcular normal al plano que contenga los vectores del hueso y la pelota: producto vectorial entre el hueso y la pelota (realizado durante el primer proceso).

$$PvecHP = (ball \rightarrow speed) \times bone \tag{3.21}$$

- Calcular normal al plano de colisión: producto vectorial entre PvecHP y bone.

$$NormalInt = PvecHP \times bone \tag{3.22}$$

(b) Normal para el caso de las skew lines:

- Calcular el punto Pd del vector director de la pelota que esta a una distancia $D = R_{pelota}$ del punto del hueso donde se ha detectado la colisión PtH. En el apartado 2.5.1.2 se explica uno de los métodos para obtener este punto pero, debido a la complejidad matemática del método explicado, se ha optado por aplicar uno más sencillo basado en el Teorema de Pitágoras. De esta forma, tomando el triángulo mostrado en la figura 3.10 que se define mediante los puntos del vector director de la pelota Pd y PtP (punto del vector de la pelota más cercano al hueso) y el punto de la recta PtH (punto de la recta más cercano al vector de la pelota) y sabiendo

$$\begin{aligned} R_{pelota} &= |\vec{D}| \quad siendo \quad D = PtH - Pd \\ DisHP &= |\vec{A}| \quad siendo \quad A = vecHPdis = PtP - PtH \\ \hat{p} &= \frac{\vec{b}}{|\vec{b}|} \quad siendo \quad b = ball \rightarrow speed \end{aligned}$$

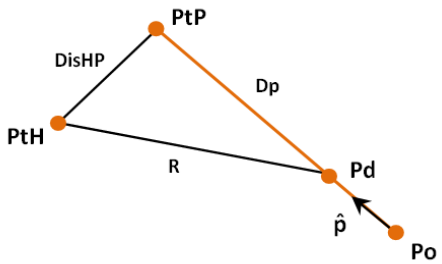


Figura 3.10: Triangulo definido por los puntos PtP, PtH y Pd.

se tiene:

$$Pd = PtP - Dp \cdot \hat{p} \quad siendo \; Dp = \sqrt{R_{pelota}^2 - D_{min}^2} \tag{3.23}$$

- Calcular el punto del hueso Q, resultante de la intersección del hueso con el plano perpendicular a este y que contiene a Pd. Ya que el plano buscado es perpendicular

al hueso, su normal coincidirá con el vector director del hueso, $\hat{N} = \frac{bone}{|bone|}$; por otro lado, la constante del plano D se calculará realizando el producto escalar de la normal calculada con el punto de la pelota obtenido al calcular la distancia mínima (PtP), de esta forma se tiene $D = -(\hat{N} \cdot PtP)$. Una vez conocido el plano, para calcular el punto Q pedido, se procede a calcular el parámetro r que, sustituido en la ecuación del hueso, determina el punto pedido.

$$r = \frac{-(D + (boneB) \cdot \hat{N})}{bone \cdot \hat{N}} \tag{3.24}$$

siendo $boneB$ el punto que se ha definido como inicial del hueso. Antes de calcular el punto buscado, se verifica que r este comprendido entre0 y 1, para asegurar que el punto obtenido mediante su aplicación sea parte del hueso.

$$Q = boneB + r \cdot bone \tag{3.25}$$

– Calcular la normal al plano de colisión: vector formado por los puntos Q y PtP.

$$NormalSkew = \frac{\vec{s}}{|\vec{s}|} \quad con \quad \vec{s} = PtP - Q \tag{3.26}$$

3. **Cálculo del nuevo vector de movimiento**
Finalmente, tras determinar que se produce colisión y habiendo calculado la normal del plano de colisión respectivo, se procede a calcular el nuevo vector de movimiento ($ball \rightarrow speed$). Este calculo se aplica en cada uno de los bucles.

Para determinar el nuevo valor del vector de movimiento, lo primero es obtener el producto escalar de este vector con la normal al plano de colisión calculada para el caso correspondiente (Proceso 2).

$$dot = (ball \rightarrow speed) \cdot \hat{N} \tag{3.27}$$

Obtenido este producto, se procede a verificar si la pelota se acerca o se aleja del objeto para el cuál se ha obtenido que se produce colisión (la cabeza, el torso o un hueso). Si el resultado es positivo ($dot > 0$) la pelota se aleja, por lo que no se produce colisión y el valor del vector de movimiento queda imperturbado. Si es negativo ($dot <= 0$) sí se produce la colisión, correspondiéndose el nuevo valor del vector con:

$$(ball \rightarrow speed) = (ball \rightarrow speed) - 2 * dot * \hat{N} \tag{3.28}$$

En caso de no haberse producido colisión para ninguno de los casos, la función retornará el valor inicial del vector de movimiento de la pelota.

A continuación se muestra el algoritmo genérico de los bucles desarrollados (3.5), con los 3 procesos descritos ya integrados. En el caso de la colisión con la cabeza, el algoritmo siguiente no ejecutaría el cálculo del punto de colisión ni su validación, de producirse colisión, pasaría directamente a calcular la

normal, como se ha explicado con anterioridad.

Data: Datos de las articulaciones de los esqueletos de los usuarios presentes, obtenidos por la función **skeltrack()**. El número de iteraciones del bucle (iter) queda definido por usersNumber para la cabeza y el torso, y por bones.size(), número total de huesos definidos, para el choque con los huesos.
Result: Determinación del valor del vector de movimiento o velocidad de la pelota (ball->speed).
for *i = 0 hasta i <iter* **do**
    Verificar las condiciones para que se pueda producir la colisión;
    if *condiciones == OK* **then**
        Determinar colisión (punto o distancia);
        Verificar la validez de la colisión (colisión ∈ área del plano o segmento);
        if *pertenencia == OK* **then**
            Calcular $\vec{N}$ al plano de colisión;
            Calcular el dot;
            Actualizar valor ball→speed;
            return ball→speed;
        else
            continue;
    else
        continue;

Algoritmo 3.5: Algoritmo generalizado de los bucles que determinan si se produce colisión con la pelota.

3.6 Cierre del sistema

Como se ha mencionado con anterioridad, esta fase es la encargada de garantizar la liberación de los recursos empleados por la librería desarrollada.

El cierre del sistema se divide en dos grandes bloques: `Cierre de la subventana` y `Cierre`, como se puede ver en el diagrama de bloques presente en 3.1, los cuáles se explican con más detalle a continuación.

3.6.1 Cierre de la ventana de visualización

La ventana de visualización de los datos muestra los datos obtenidos por la kinect y esta configurada de tal modo que se puede solicitar su creación y destrucción en cualquier momento de la ejecución de la aplicación. En este apartado se va a proceder a explicar en qué consiste su cierre, de igual forma que en el apartado 3.3 se ha explicado su creación y manejo.

El cierre de la ventana de visualización consiste en la liberación de los recursos asociados a ella y su correspondiente preparación para el caso de que se vuelva a solicitar su creación. Por ello, este fase se encarga de destruir la ventana creada mediante la llamada a la función `glutDestroyWindow()`, a la cuál se le debe pasar como argumento el identificador asociado a la ventana. Una vez destruida, se procede a la liberación de la memoria reservada para la generación de la textura, que almacenaba los datos obtenidos por la cámara. Finalmente, se realiza la actualización a los valores iniciales de las variables

creadas para verificar su existencia y operar en función de esta. Esta funcionalidad se implementa en la función `_closeSubWindow()`.

3.6.2 Cierre de la librería

El cierre de la librería se realiza cuando se solicita el cierre de la aplicación, y se implementa a través de la función `_closure()`.

Al igual que en el caso del cierre de la ventana de datos, esta fase se encarga de la liberación de los recursos utilizados por la librería. Así, esta función procede de la siguiente forma: primero libera los recursos utilizados por la ventana de visualización de datos, en el caso de que existiera, a través de la llamada de la función mencionada en el apartado anterior; después, procede a la liberación de la memoria reservada para los vectores creados para el almacenamiento de los datos identificativos de los usuarios; y, finalmente, libera los objetos asociados al software de OpenNI y NiTE y realiza el cierre de ambos software.

Para este último paso, se realiza la llamada a los destructores de cada uno de esos objetos implementados en sus respectivas librerías, siendo estos objetos aquellos que pertenecen a las clases `nite::UserTracker`, `openni::VideoStream` y `openni::Device`, para finalmente realizar la llamada a las funciones `shutdown`, asociadas a cada librería.

Capítulo 4

Resultados

Cuanto más grande es la dificultad, más gloria hay en superarla.

Epícuro

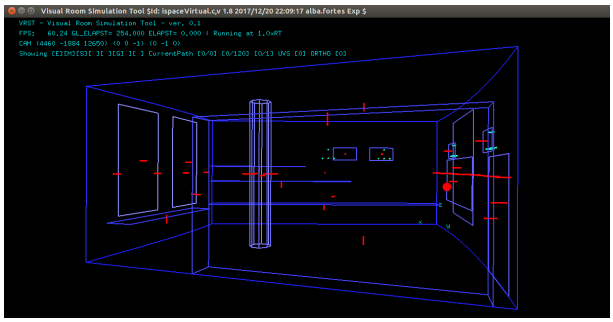
4.1 Introducción

En este capítulo se procede a presentar los resultados obtenidos con la ejecución del algoritmo diseñado para la cumplimentación de este TFG. En primer lugar, se proceden a explicar los entornos virtuales de los que se ha hecho uso para su desarrollo y verificación, junto con una descripción de la ubicación de la cámara en ellos. Para, a continuación, proceder a la exposición de los resultados obtenidos.

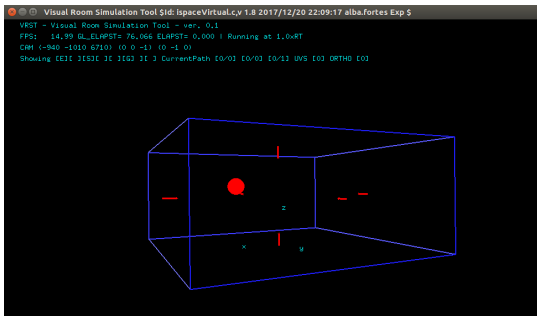
4.2 Entorno experimental

Este trabajo se ha realizado en un entorno cerrado, específicamente en el aula de laboratorio del Grupo de ingeniería electrónica aplicada a Espacios Inteligentes y Transporte (GEINTRA).

Como se ha indicado con anterioridad, el algoritmo desarrollado se ejecuta sobre un entorno virtual previamente creado, mostrado en la figura 4.1a, que refleja el espacio del laboratorio donde se ha realizado el proyecto. El archivo que contiene la información de este entorno es `ispace.sim`. Además de este entorno, debido a la gran diferencia de tamaño entre el espacio del entorno del laboratorio y el espacio delimitado por el campo visual de la kinect v2, se ha hecho uso de un entorno reducido para facilitar la verificación del algoritmo de colisión desarrollado, ya que al tener un menor espacio de movimiento es más probable que la pelota colisione contra los usuarios detectados. El archivo que contiene los datos de este entorno, mostrado en la figura 4.1b, es `idiapRoom-AV163-onlyWalls.sim`.



(a) laboratorio GEINTRA.



(b) verificación de la colisión.

Figura 4.1: Entornos utilizados para el desarrollo del sistema creado.

Por último, debido al método aplicado en el desarrollo del algoritmo, la cámara kinect puede ser ubicada en cualquier posición de este espacio, con la única condición de que se deben indicar los datos relativos a su nueva posición y orientación de forma previa a la compilación y ejecución del código. La forma de realizar este cambio se indica en el apéndice *Manual de usuario* (D.3).

Inicialmente, la kinect ha sido ubicada a 6 metros en la dirección del eje X, 0.48 metros en la del eje Y e 0.64 metros en la del eje Z, siendo estos ejes los del entorno virtual. Así mismo, su orientación es paralela al suelo y con su eje X paralelo y opuesto al eje X del entorno, y su eje Z paralelo y en el mismo sentido que el eje Y del entorno. En la figura 4.2, se muestra la ventana resultante de la ejecución del algoritmo con la cámara en la posición descrita.

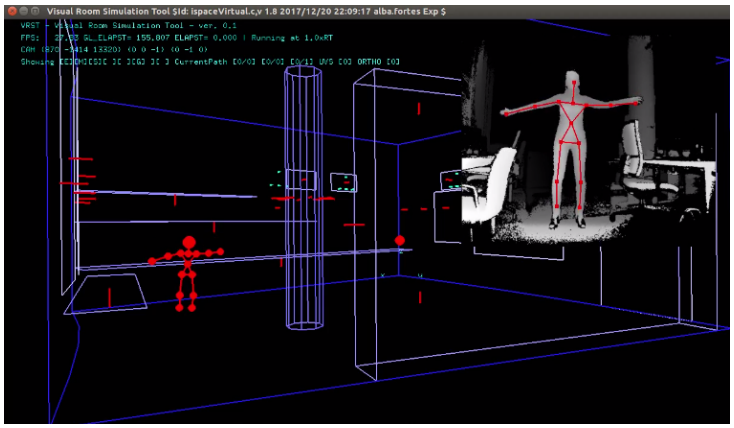


Figura 4.2: Resultado de ejecutar el algoritmo con la kinect en la posición inicial.

Se debe tener en cuenta que al cambiar de entorno, también cambia el sistema de referencia, por lo que para que el algoritmo conserve la ubicación de la cámara de la forma descrita para el primer entorno (`ispace.sim`), se deben alterar los parámetros que la configuran para que se adapten al SR del nuevo entorno. De esta forma, la nueva ubicación relativa de la cámara sería:

- Posición: a 0.05 metros en la dirección del eje X, 1.5 metros en la del eje Y e 0.5 metros en la del Z.
- Orientación: el eje X en el sentido opuesto al del entorno, el eje Y paralelo al eje Z del entorno y el eje Z paralelo al eje Y del entorno.

En la figura 4.3, se muestra la ventana resultante de la ejecución del algoritmo con la cámara en la posición descrita, al igual que se ha hecho para el entorno anterior.

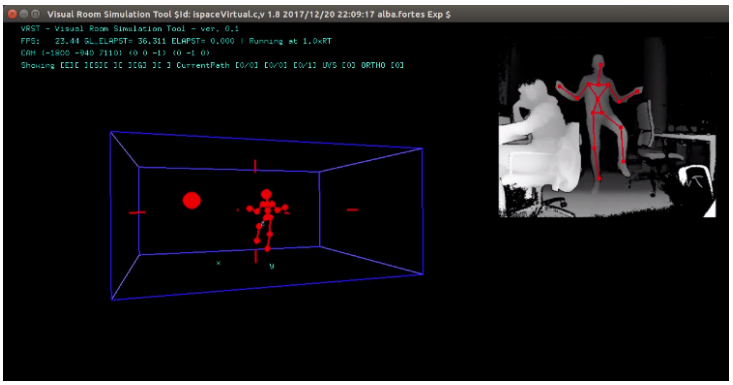
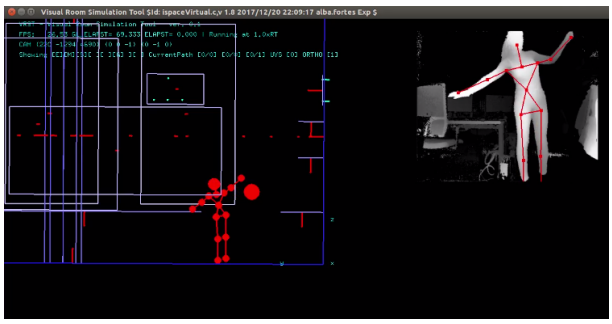


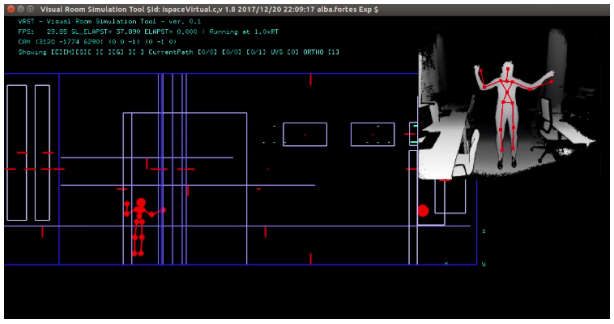
Figura 4.3: Resultado de ejecutar el algoritmo con el entorno creado para verificar la colisión.

En la figura 4.4, mostrada a continuación, se presentan múltiples posiciones de la cámara, como método de verificación de que el traspaso de coordenadas entre la kinect y el entorno virtual se da correctamente sin importar donde se haya ubicado la cámara. Las posiciones mencionadas se corresponden con:

- **Posición 2:** el origen de la kinect esta ubicado en el punto (6.80 , 1.43, 0.64) dado en metros y se encuentra orientada de forma que sus ejes quedan, respecto a los del entorno, $x_k = -y_e$, $y_k = z_e$ y $z_k = -x_e$.
- **Posición 3:** el origen de la kinect esta ubicado en el punto (6 , 0.315, 1.40) dado en metros y se encuentra orientada de forma que sus ejes quedan, respecto a los del entorno, $x_k = -x_e$, y el plano YZ de la kinect paralelo al del entorno y girado 70º en sentido antihorario respecto al eje X del entorno.



(a) Posición 2.



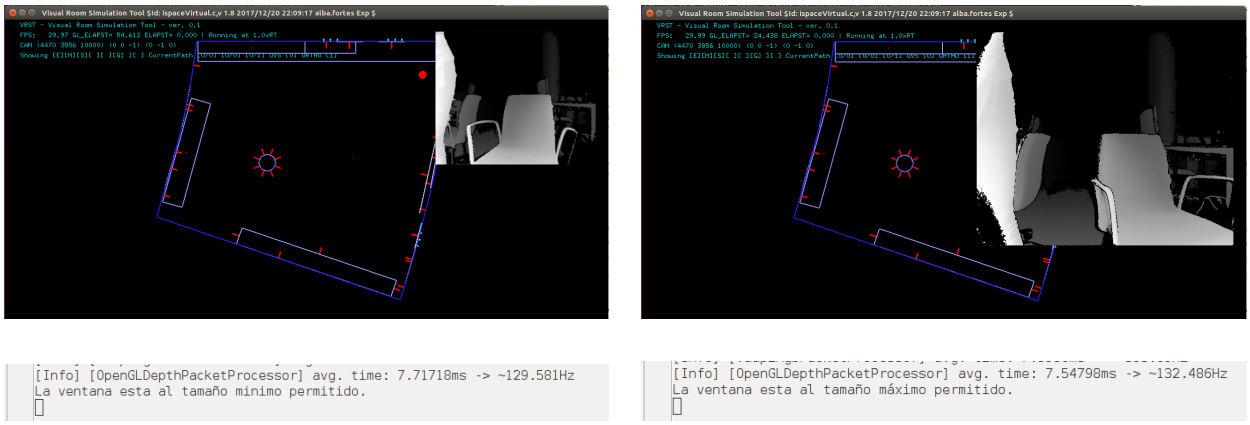
(b) Posición 3.

Figura 4.4: Verificación de la modificación de la ubicación de la kinect en el entorno.

4.3 Resultados experimentales

En este apartado se presentan los resultados obtenidos con el algoritmo desarrollado.

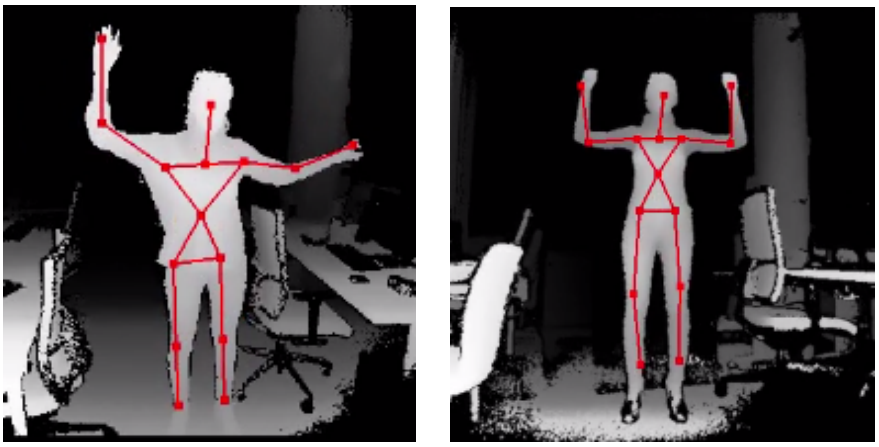
En primer lugar, se ha realizado la verificación de que la ventana de visualización cumpla con las restricciones impuestas. Como se explico en 3.3, la ventana de visualización de datos se ha configurado de tal modo que su tamaño se vea limitado a un máximo y a un mínimo, siendo estos (512x424) y (320x265) respectivamente. Esta limitación se ve reflejada en la figura 4.5, donde se muestran ambos casos junto con el mensaje de aviso recibido al alcanzarlos.



(a) Mínimo alcanzado. (b) Máximo alcanzado.

Figura 4.5: Ejemplo de la limitación del tamaño de la ventana de visualización de datos.

El siguiente punto a verificar, antes de proceder al análisis del cumplimiento de los objetivos, es que la calibración del esqueleto se pueda realizar mediante la calibración automática o mediante la identificación de pose (PSI). En la figura 4.6 se muestra un ejemplo de ambos casos.



(a) Autocalibración. (b) Calibración por pose PSI.

Figura 4.6: Muestra de modos de calibración implementados.

El algoritmo implementado permite la identificación y seguimiento de hasta 6 usuarios. Por ello, a continuación se mostrarán los resultados obtenidos para ambos casos, un único usuario o múltiples usuarios, donde los datos de las tasas de confianza mostrados se han obtenido de la forma mencionada al final del apartado 3.4.

- **1 usuario**

  En la figura 4.7, se muestran varios ejemplos de la identificación y seguimiento de un usuario en la escena. Como se puede ver, el algoritmo implementado es capaz de ubicar al usuario en la posición correcta, representando su esqueleto de la forma determinada en 3.4. Para determinar la precisión de estas medidas se ha obtenido la tasa de confianza de los datos de los esqueletos en múltiples frames, dada en un rango de 0 a 1. Estos datos se representan en las tablas 4.1 y 4.2 donde cada una de sus columnas representan los datos obtenidos en distintas frames.

  Como se puede ver, en general se obtienen resultados de alta precisión, llegando a alcanzar el 94 % de confianza en múltiples casos, siendo las articulaciones que conforman el torso del esqueleto (hombros,

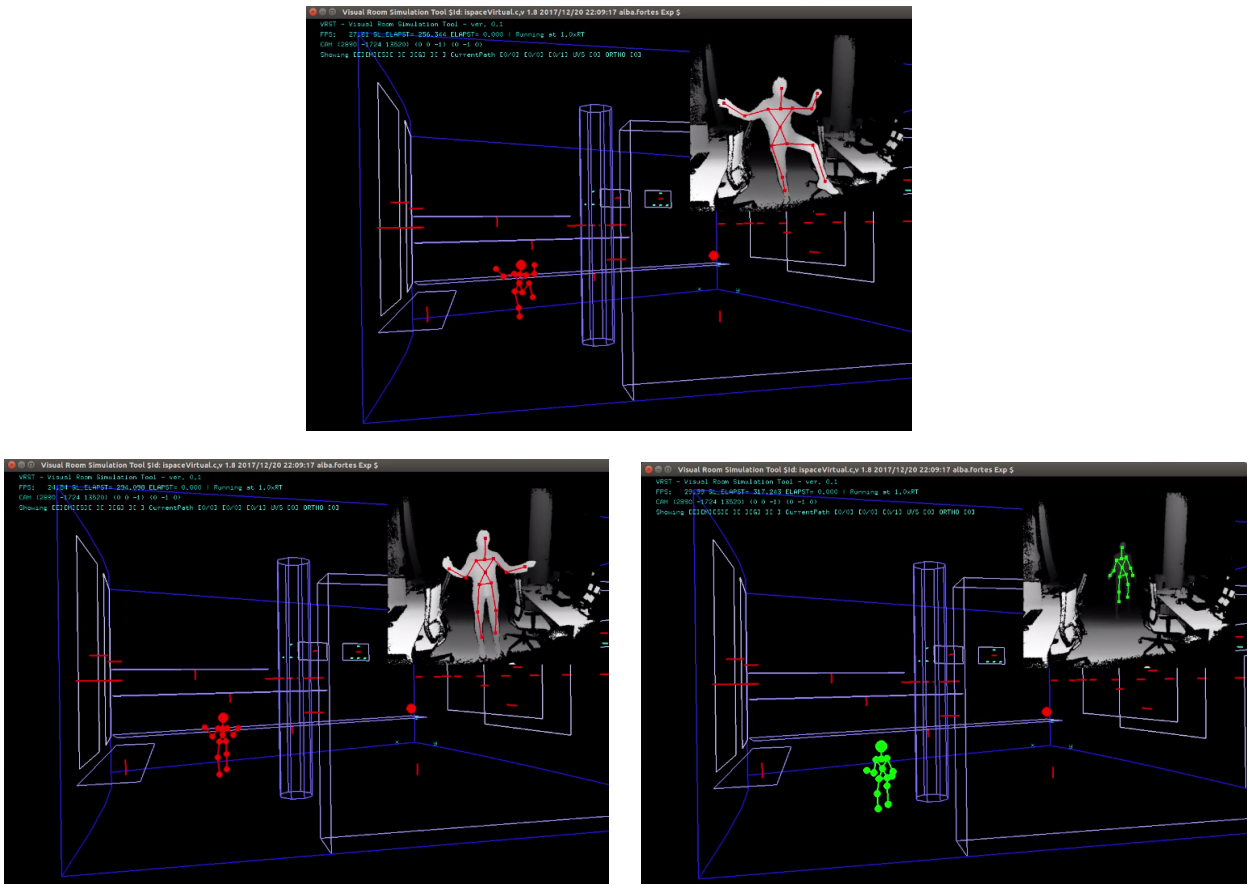
torso y caderas) las que obtienen resultados más precisos y estables. Comparando los datos de posición y orientación se observa un claro decremento en la calidad de los datos de orientación, llegando a alcanzar el 0 % de confianza para las extremidades (manos y pies), aún cuando su posición presenta buenos resultados.

Articulación	Muestra 1	Muestra 2	Muestra 3	Muestra 4	Muestra 5	Media T.C.
CABEZA	0.700	1.000	1.000	1.000	0.000	0.740
CUELLO	0.700	1.000	1.000	1.000	1.000	0.940
HOMBRO D.	0.700	1.000	1.000	1.000	1.000	0.940
HOMBRO I.	0.700	1.000	1.000	1.000	1.000	0.940
CODO D.	0.700	0.500	1.000	0.500	0.500	0.640
CODO I.	0.700	1.000	1.000	1.000	0.500	0.840
MANO D.	0.700	0.500	1.000	0.500	0.500	0.640
MANO I.	0.700	1.000	1.000	1.000	0.500	0.840
TORSO	0.700	1.000	1.000	1.000	1.000	0.940
CADERA D.	0.700	1.000	1.000	1.000	1.000	0.940
CADERA I.	0.700	1.000	1.000	1.000	1.000	0.940
RODILLA D.	0.700	0.500	0.500	1.000	1.000	0.740
RODILLA I.	0.700	1.000	1.000	1.000	1.000	0.940
PIE D.	0.700	0.500	0.500	1.000	0.000	0.540
PIE I.	0.700	1.000	1.000	1.000	1.000	0.940

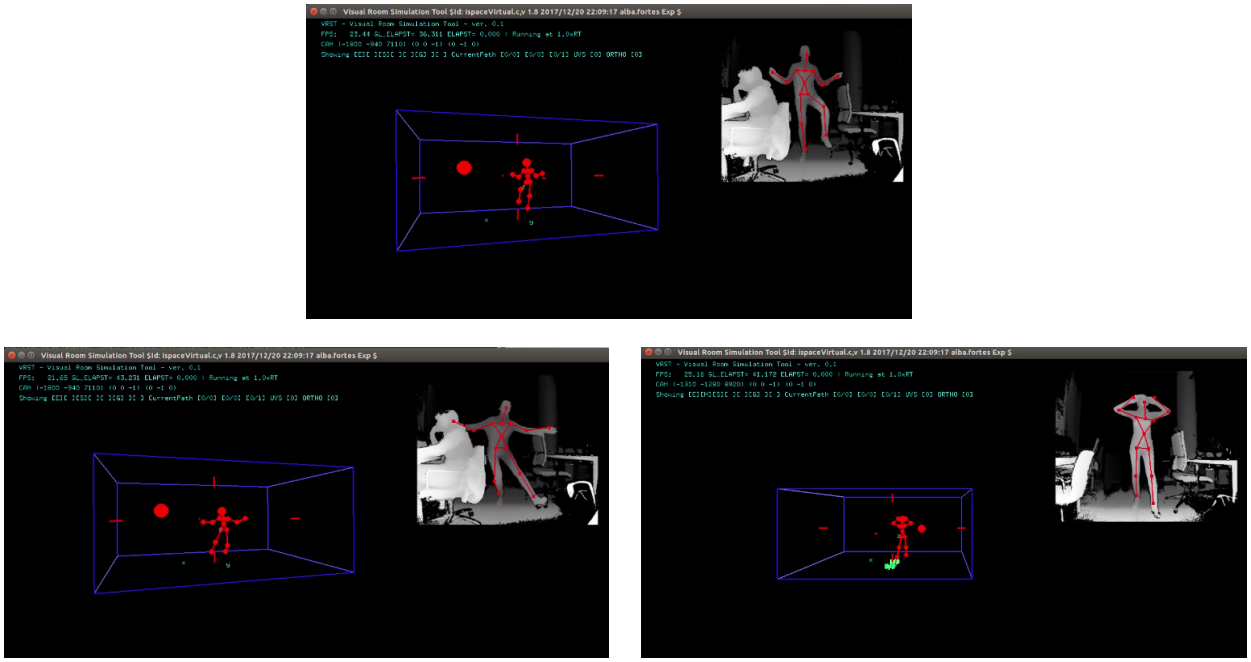
Tabla 4.1: Tasas de confianza de los datos de posición de las articulaciones.

Articulación	Muestra 1	Muestra 2	Muestra 3	Muestra 4	Muestra 5	Media T.C.
CABEZA	0.700	1.000	1.000	1.000	0.000	0.740
CUELLO	0.700	1.000	1.000	1.000	1.000	0.940
HOMBRO D.	0.700	0.500	1.000	0.500	0.500	0.640
HOMBRO I.	0.700	1.000	1.000	1.000	0.500	0.840
CODO D.	0.700	0.500	1.000	0.500	0.500	0.640
CODO I.	0.700	1.000	1.000	1.000	0.500	0.840
MANO D.	0.000	0.000	0.000	0.000	0.000	0.000
MANO I.	0.000	0.000	0.000	0.000	0.000	0.000
TORSO	0.700	1.000	1.000	1.000	1.000	0.940
CADERA D.	0.700	0.500	0.500	1.000	1.000	0.740
CADERA I.	0.700	1.000	1.000	1.000	1.000	0.940
RODILLA D.	0.700	0.500	0.500	1.000	0.000	0.540
RODILLA I.	0.700	1.000	1.000	1.000	1.000	0.940
PIE D.	0.000	0.000	0.000	0.000	0.000	0.000
PIE I.	0.000	0.000	0.000	0.000	0.000	0.000

Tabla 4.2: Tasas de confianza de los datos de orientación de las articulaciones.



(a) en entorno GEINTRA.



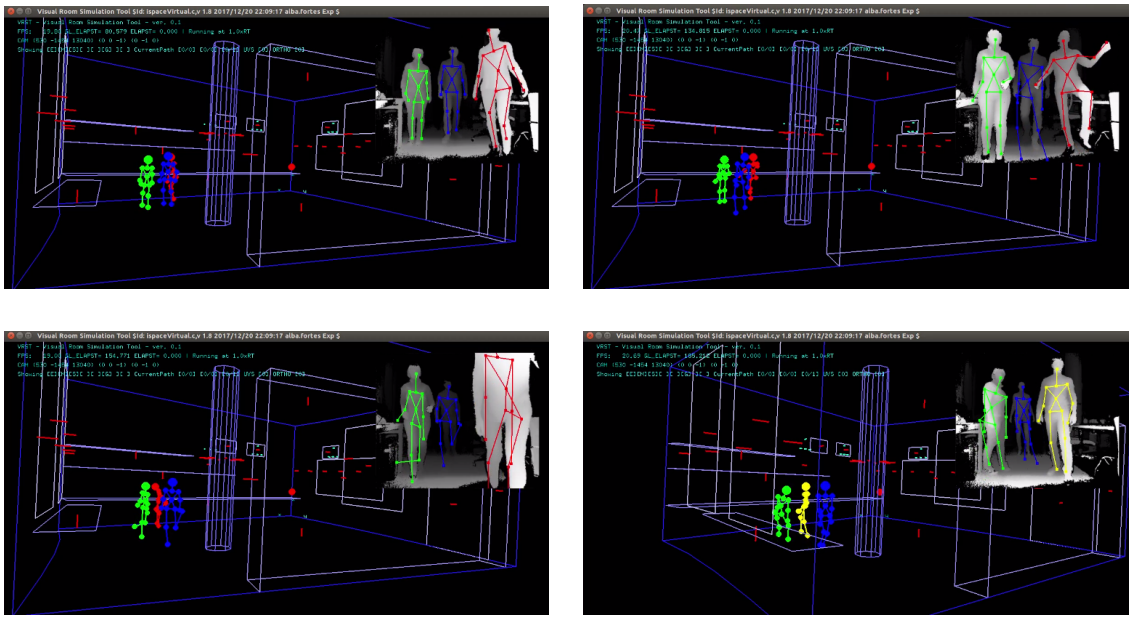
(b) en entorno colisión.

Figura 4.7: Muestras de identificación y seguimiento de un usuario extraídas de múltiples frames.

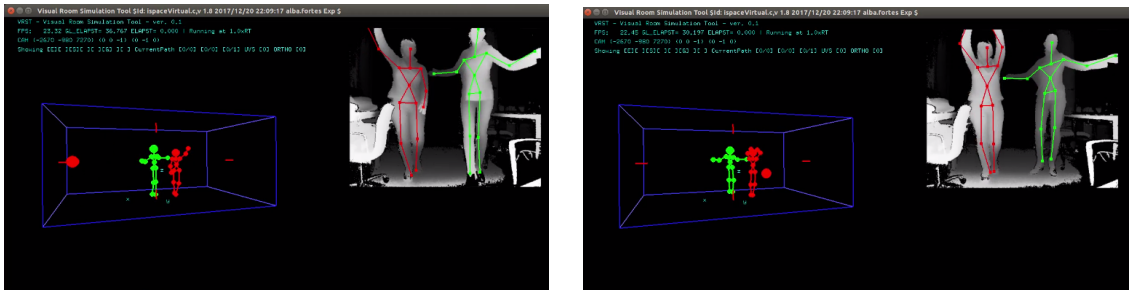
- **Múltiples usuarios**

  A continuación, en la figura 4.8, se muestran varios ejemplos de la identificación y seguimiento en el caso de que haya más de un usuario en la escena. Como se puede ver, al igual que con el caso de un único usuario, el algoritmo implementado es capaz de identificar y realizar el seguimiento del esqueleto de múltiples usuarios, incluso cuando uno de los usuarios bloquea la visual de otro de ellos. Como se hizo en el caso anterior, se han obtenido las tasas de confianza de las medidas obtenidas por la kinect para poder determinar la calidad de estas. En las tablas 4.3 y 4.4 se muestran las medias de las tasas de confianza obtenidas para cada articulación de todos los usuarios presentes en una misma frame y, después, se calcula la media de estas para obtener el valor más representativo posible.

  Comparando este caso con el de un único usuario se puede observar un descenso en la precisión de las medidas, siendo la máxima obtenida del 89 %. Uno de los motivos de este descenso es la presencia de oclusiones provocadas por la superposición de los usuarios. En este caso también se detecta la pauta de que los datos de orientación presenten menor fiabilidad que los de posición, ya mencionada cuando solo había un usuario y, que al igual que en ese caso, resulta en que la tasa de confianza de la orientación de las extremidades también alcance el 0 %.



(a) en entorno GEINTRA.



(b) en entorno colisión.

Figura 4.8: Muestras de identificación y seguimiento de varios usuarios extraídas de múltiples frames.

Articulación	Media 1	Media 2	Media 3	Media total
CABEZA	0.75	0.71	0.9	0.79
CUELLO	0.86	0.9	0.9	0.89
HOMBRO D.	0.86	0.9	0.9	0.89
HOMBRO I.	0.86	0.9	0.9	0.89
CODO D.	0.64	0.48	0.56	0.56
CODO I.	0.7	0.58	0.65	0.64
MANO D.	0.64	0.48	0.62	0.58
MANO I.	0.7	0.5	0.5	0.56
TORSO	0.86	0.9	0.9	0.89
CADERA D.	0.86	0.9	0.9	0.89
CADERA I.	0.86	0.9	0.9	0.89
RODILLA D.	0.7	0.73	0.73	0.72
RODILLA I.	0.64	0.67	0.73	0.68
PIE D.	0.48	0.62	0.73	0.61
PIE I.	0.64	0.45	0.4	0.5

Tabla 4.3: Medias de las tasas de confianza de los datos de posición de las articulaciones en el caso de múltiples usuarios.

Articulación	Media 1	Media 2	Media 3	Media total
CABEZA	0.75	0.71	0.9	0.79
CUELLO	0.86	0.9	0.9	0.89
HOMBRO D.	0.64	0.48	0.56	0.58
HOMBRO I.	0.7	0.58	0.65	0.64
CODO D.	0.64	0.48	0.56	0.56
CODO I.	0.7	0.5	0.5	0.56
MANO D.	0	0	0	0
MANO I.	0	0	0	0
TORSO	0.86	0.9	0.9	0.89
CADERA D.	0.7	0.73	0.73	0.72
CADERA I.	0.64	0.67	0.73	0.68
RODILLA D.	0.48	0.62	0.73	0.61
RODILLA I.	0.64	0.45	0.4	0.5
PIE D.	0	0	0	0
PIE I.	0	0	0	0

Tabla 4.4: Medias de las tasas de confianza de los datos de orientación de las articulaciones en el caso de múltiples usuarios.

Otro factor a verificar es la eficacia de la detección de la colisión entre la pelota y los usuarios presentes. Para empezar a analizar los resultados obtenidos de las colisiones, se debe tener en cuenta que la probabilidad de que se produzca la colisión de 2 objetos en el espacio es matemáticamente baja. Partiendo de este punto, en las figuras 4.9 y 4.10, se muestran varios ejemplos de casos de colisión producidos al ejecutar el código implementado.

Observando el algoritmo en ejecución, se puede afirmar que el algoritmo desarrollado cumple con su función de generar una interacción entre el usuario y los componentes virtuales, en este caso con una pelota que se ve desviada al colisionar contra el esqueleto. La observación de los resultados también permite apreciar que el algoritmo desarrollado presenta múltiples errores en la detección, siendo el caso desarrollado para la detección de las skew lines el más ineficiente, lo que se explica en mayor detalle más adelante.

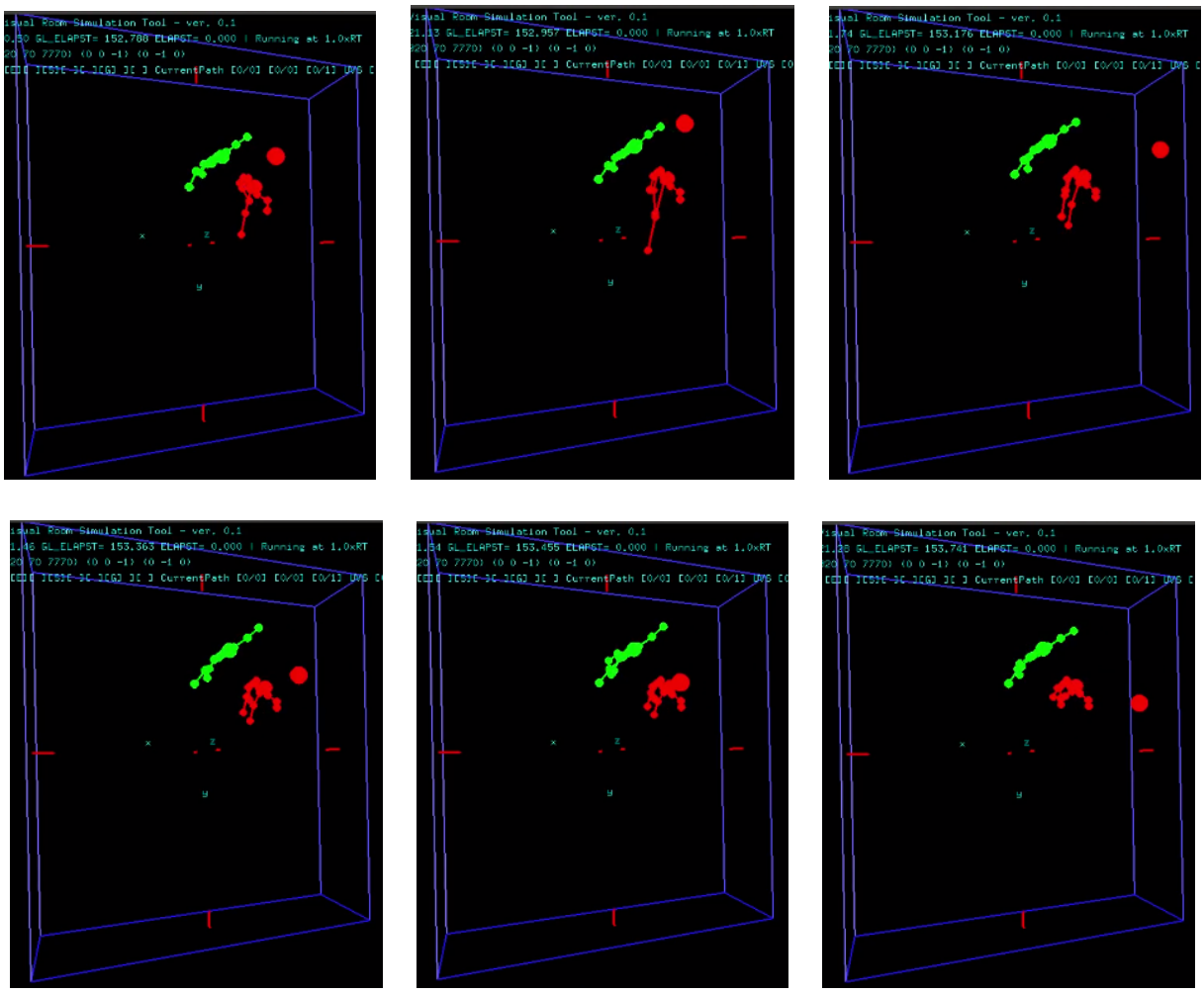
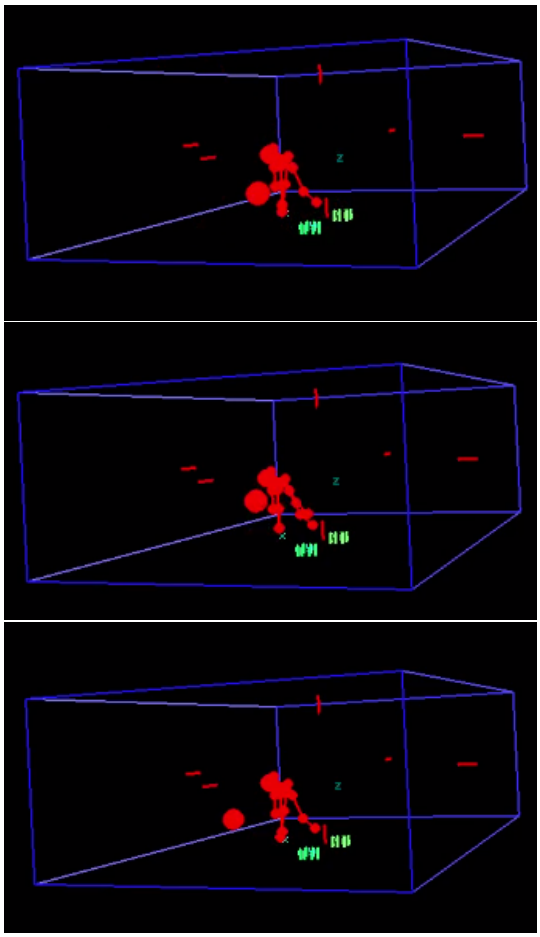
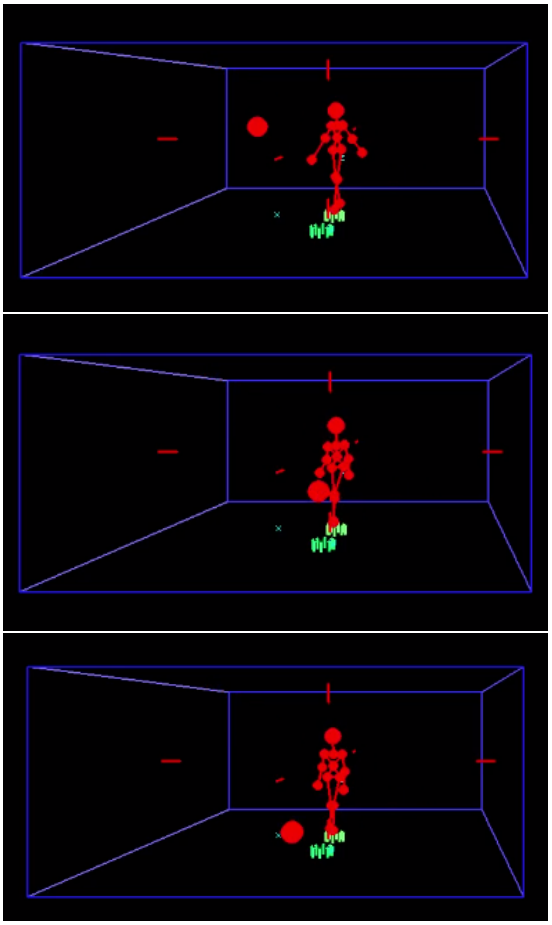


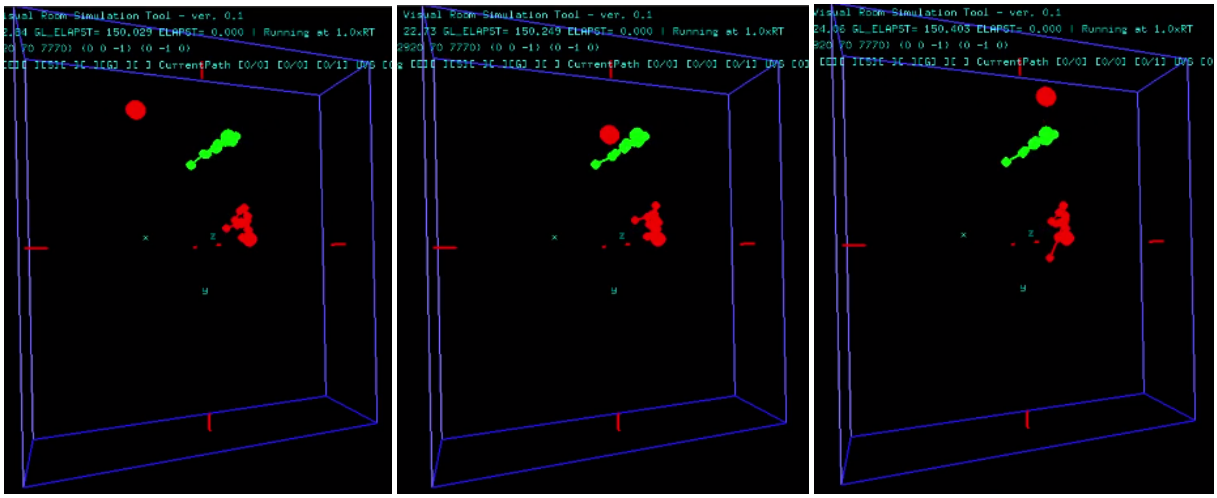
Figura 4.10: Ejemplo de colisión cuando hay más de un usuario (choque contra ambos usuarios).



(a) entre la cabeza y la pelota.



(b) entre un hueso y la pelota.



(c) con el torso.

Figura 4.9: Ejemplos de colisiones producidas durante la ejecución.

Para terminar, a continuación se exponen los errores más importantes que se han detectado durante la ejecución:

1. **Falsos positivos en la detección de usuarios**
   En ocasiones la kinect v2 identifica objetos como usuarios, siendo el caso más común la detección de la columna presente en la escena como si fuera un usuario; en otras ocasiones, produce un error por el cual lee a un usuario más de una vez, generando errores en la representación de los datos como se muestra en la figura 4.11.

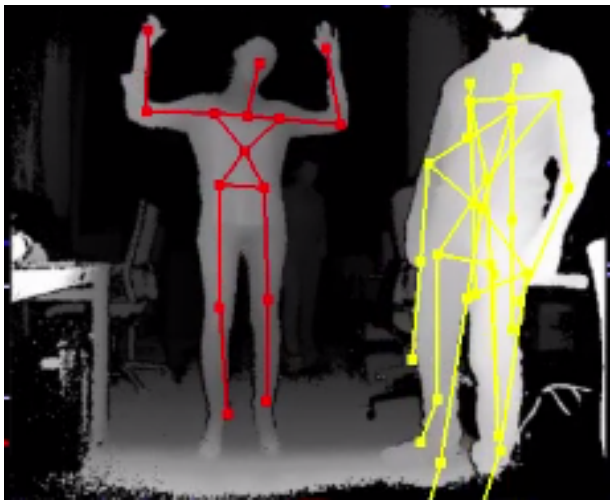
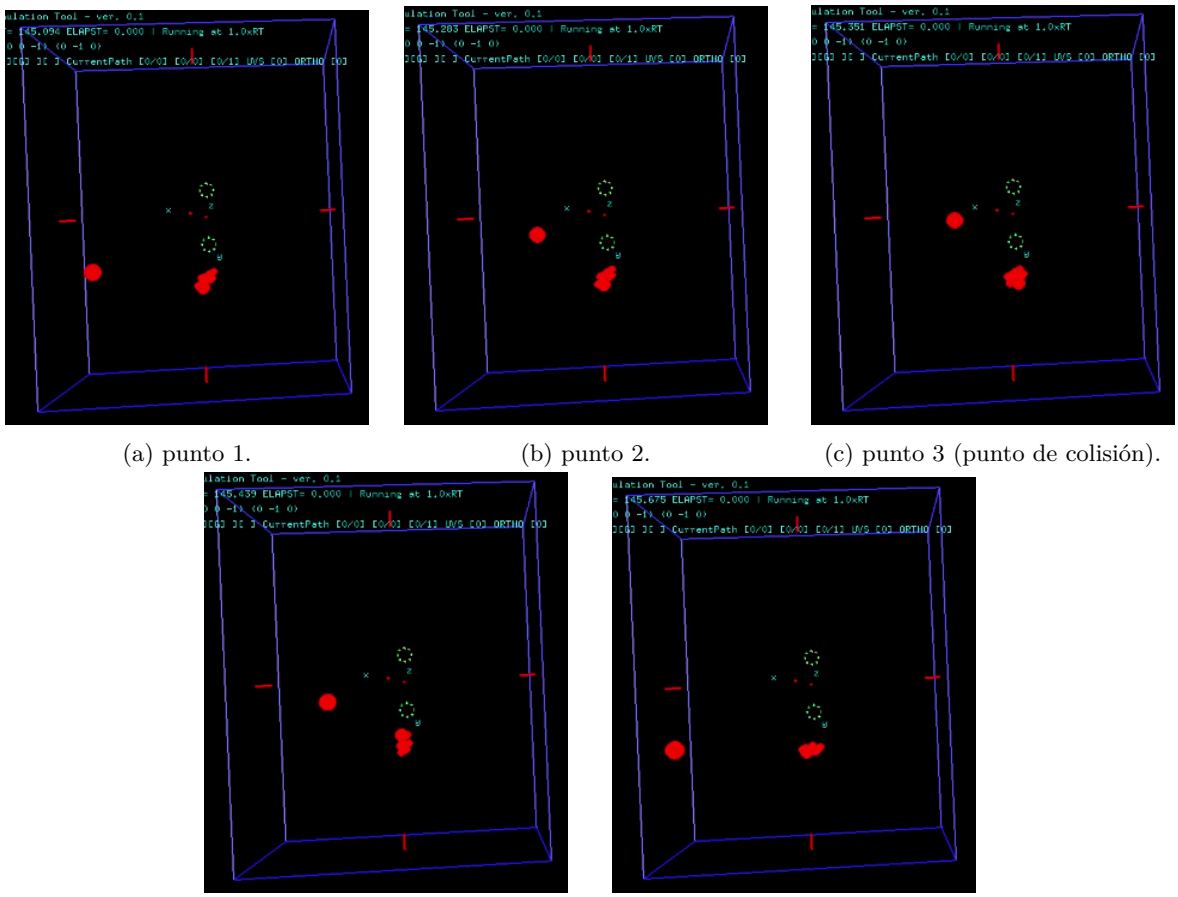


Figura 4.11: Error: Identificación de un usuario múltiples veces.

2. **Inestabilidad de los esqueletos dibujados**
   Al ejecutar el código, se puede observar que los datos dados de las articulaciones (especialmente los de las piernas) no son correctos o no logran seguir el movimiento realizado, un error que es introducido por la propia kinect, ya que es esta la que proporciona los datos empleados por el algoritmo.

3. **Errores en el algoritmo de colisión**
   La ejecución del algoritmo demuestra que aunque el objetivo buscado se alcanza, el método implementado para identificar las colisiones no es eficiente, especialmente para el caso de las skew lines. Esto se debe a que en muchas ocasiones, acercamientos entre la pelota y el usuario que deberían considerarse choques no son detectados debido a la precisión buscada por el algoritmo implementado, mientras que al mismo tiempo se producen colisiones (como la mostrada en la figura 4.12) que representan claros falsos positivos.



(a) punto 1. (b) punto 2. (c) punto 3 (punto de colisión).

(d) punto 4. (e) punto 5.

Figura 4.12: Ejemplo de un falso positivo detectado por el algoritmo de colisión.

Capítulo 5

Conclusiones y líneas futuras

En este apartado se exponen las conclusiones obtenidas tras su realización y se plantean posibles líneas de desarrollo a seguir en el futuro.

5.1 Conclusiones

En este trabajo se ha implementado una herramienta de visualización e interacción virtual con los usuarios captados por una cámara ToF kinect v2.

Para ello se ha creado un algoritmo, dividido en 2 procesos, un proceso principal de detección, representación e interacción con los usuarios y un proceso secundario de visualización, a partir de los cuáles se alcanza el objetivo buscado. Además, debe señalarse, que para este trabajo solo se han hecho uso de los datos obtenidos por el sensor de profundidad de la cámara utilizada, lo que permite conservar la privacidad de las personas detectadas y, habilita la utilización de la librería desarrollada para sistemas ubicados en zonas con restricciones debidas a la ley de protección de datos.

Para la obtención de los datos referentes a las articulaciones que definen los esqueletos, se han requerido las librerías OpenNI y NiTE. La combinación de estas librerías realiza el procesamiento de los datos obtenidos por la kinect que permite obtener datos útiles para el programador, como en este caso son las posiciones y orientaciones de cada una de las articulaciones.

La evaluación del algoritmo implementado se ha llevado a cabo mediante el análisis y la observación de los resultados obtenidos mediante su ejecución, tal y como se muestra en el capítulo 4.

En base a los resultados obtenidos en el capítulo mencionado, se puede determinar que las fases de visualización, identificación y representación alcanzan los objetivos deseados con una tasa de error aceptable, siendo este error generado por la cámara, lo que los convierte en errores aleatorios sobre los que no se tiene efecto en lo referente al algoritmo implementado. Por otro lado, la fase de colisión , aunque cumple con el objetivo buscado al permitir la interacción del usuario con elementos virtuales, presenta un tasa de error bastante elevada, provocada principalmente por los errores introducidos por el método matemático implementado, por lo que se considera que la implementación y prueba de otros métodos podría obtener resultados más óptimos que los alcanzados.

En resumen, se puede concluir que aunque hay margen de mejora, los objetivos perseguidos se han alcanzado con éxito.

5.2 Líneas futuras

En este apartado se realizan una serie de propuestas sobre posibles líneas de investigación a seguir en trabajos futuros.

- **Mejora del algoritmo de representación.** Como se ha podido ver durante la realización de este proyecto, aunque el algoritmo desarrollado cumple con la funcionalidad buscada, aún hay ciertos factores, como la estabilidad de los esqueletos representados, que se pueden mejorar. Mejora que permitirá entre otras cosas la reducción de la aparición de falsos positivos.
- **Implementación de otros métodos de detección de la colisión.** Debido a la necesidad de precisión matemática del algoritmo desarrollado, la respuesta obtenido no alcanza el resultado deseado. Por ello, se plantea la búsqueda e integración de otros métodos, con el objetivo de lograr una interacción más precisa y fluida.
- **Mejora general del algoritmo.** La mejora de diversos aspectos del código permitiría la generación de un sistema mucho más atractivo para sus usuarios, así como ampliaría sus posibles aplicaciones en otros campos. Algunas de estas mejoras podrían ser la implementación en paralelo (mediante el empleo de múltiples hilos) del algoritmo como método para reducir el tiempo de respuesta o la integración de más objetos interactivos y la asociación de estos a diversas respuestas (cambio de posición o color del objeto, etc.) que integren una mayor funcionalidad al código.
- **Estudio del empleo de múltiples Kinect.** El empleo de 2 o más kinect, ubicadas en puntos diferentes de la escena permitiría la ampliación del campo de visión, así como reduciría las pérdidas de datos por oclusión y mejoraría la precisión del posicionamiento de las articulaciones.
- **Integración de la librería como método de control.** Se plantea el estudio de la viabilidad de la integración de la librería desarrollada (con los cambios pertinentes) en el sistema de control remoto de un mecanismo físico, por ejemplo un robot o algún mecanismo móvil.

Bibliografía

[1] S. T. López, "Mejoras en un simulador virtual de espacio inteligente. trabajo fin de grado," 2017.

[2] D. Casillas Pérez, "Diseño, implementación y evaluación de una interfaz de control multimodal en un espacio inteligente: control gestual," Master's thesis, Escuela Politécnica Superior. Universidad de Alcalá. Spain, 2013.

[3] E. Lachat, H. Macher, T. Landes, and P. Grussenmeyer, "Assessment and calibration of a rgb-d camera (kinect v2 sensor) towards a potential use for close-range 3d modeling," *Remote Sensing*, vol. 7, no. 10, pp. 13 070–13 097, 2015. [Online]. Available: http://www.mdpi.com/2072-4292/7/10/13070

[4] "OpenNI, información disponible en wikipedia." https://en.wikipedia.org/wiki/OpenNI [Último acceso 29/01/2018].

[5] "Página oficial de OpenNI (v2.2.0.33) y NiTE (v2.0), con acceso a versiones más antiguas y documentación." http://openni.ru/index.html [Último acceso 26/Enero/2018].

[6] *NITE Controls User Guide*, PrimeSense, documentación obtenible con la descarga del middleware NiTE v1.5.2.23.

[7] V. Villena Martínez, "Análisis comparativo de métodos de calibrado para sensores rgb-d y su influencia en el registro de múltiples vistas," Trabajo de Fin de Grado, Universidad de Alicante, Departamento de Tecnología Informática y Computación., 2015, disponible en: http://hdl.handle.net/10045/48745 [Último acceso 28/Enero/2018].

[8] O. C. Miles Hansard, Seungkyu Lee and R. P. Horaud, *Time-of-Flight Cameras: Principles, Methods and Applications*, ser. SpringerBriefs in Computer Science. Springer-Verlag London, 2012, disponible en https://hal.inria.fr/hal-00725654/PDF/TOF.pdf.

[9] *Kinect 2.0: Human Interface Guidelines*, Microsoft Corporation, documentación obtenible desde el enlace:http://download.microsoft.com/download/6/7/6/676611B4-1982-47A4-A42E-4CF84E1095A8/KinectHIG.2.0.pdf[Último acceso: 29/01/2018].

[10] J. Sell and P. O'Connor, "The xbox one system on a chip and kinect sensor," *IEEE Micro*, vol. 34, no. 2, pp. 44–53, Mar 2014.

[11] "Opengl," https://www.opengl.org [Último acceso: 31/Enero/2018].

[12] S. Briot and W. Khalil, *Homogeneous Transformation Matrix*. Cham: Springer International Publishing, 2015, pp. 19–32. [Online]. Available: https://doi.org/10.1007/978-3-319-19788-3_2

[13] D. Cross, *Fundamentals of Ray Tracing*. Don Cross, 2013, ch. 10. Mirror Reflection, disponible online en http://cosinekitty.com/raytrace/[Último acceso: 8/Febrero/2018].

[14] R. A. El-laithy, J. Huang, and M. Yeh, "Study on the use of microsoft kinect for robotics applications," in *Proceedings of the 2012 IEEE/ION Position, Location and Navigation Symposium*, April 2012, pp. 1280–1288.

[15] J. R. Ruiz-Sarmiento, C. Galindo, and J. Gonzalez, "Improving human face detection through tof cameras for ambient intelligence applications," in *Ambient Intelligence - Software and Applications*, P. Novais, D. Preuveneers, and J. M. Corchado, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 125–132.

[16] S. Fuchs, "Multipath interference compensation in time-of-flight camera images," in *2010 20th International Conference on Pattern Recognition*, Aug 2010, pp. 3583–3586.

[17] S. Lee, B. Kang, J. D.K. Kim, and C. Y. Kim, "Motion blur-free time-of-flight range sensor," vol. 8298, pp. 28–, 02 2012.

[18] M. V. den Bergh and L. V. Gool, "Combining rgb and tof cameras for real-time 3d hand gesture interaction," in *2011 IEEE Workshop on Applications of Computer Vision (WACV)*, Jan 2011, pp. 66–72.

[19] T. Köhler, S. Haase, S. Bauer, J. Wasza, T. Kilgus, L. Maier-Hein, H. Feußner, and J. Hornegger, "Tof meets rgb: Novel multi-sensor super-resolution for hybrid 3-d endoscopy," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013*, K. Mori, I. Sakuma, Y. Sato, C. Barillot, and N. Navab, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 139–146.

[20] H. Sarbolandi, D. Lefloch, and A. Kolb, "Kinect range sensing: Structured-light versus time-of-flight kinect," *Computer Vision and Image Understanding*, vol. 139, pp. 1 – 20, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077314215001071

[21] "Especificaciones de la kinect v1," disponible en https://msdn.microsoft.com/library/jj131033.aspx[Último acceso: 29/01/2018].

[22] "Página de wikipedia de kinect v1," https://en.wikipedia.org/wiki/Kinect [Último acceso: 28/01/2018].

[23] "Características de la kinect v2," disponible en https://developer.microsoft.com/es-es/windows/kinect/hardware[Último acceso: 29/01/2018].

[24] *OpenNI Programmer's Guide*, disponible en http://openni.ru/openni-programmers-guide/index.html[Último acceso: 29/Enero/2018].

[25] "Openni, alternativas de descarga (v2.2.0.33)," https://structure.io/openni [Último acceso 26/Enero/2018].

[26] "Openni, guía de referencia." http://openni.ru/reference-guide/index.html?t=index.html [Último acceso 29/Enero/2018].

[27] E. Weisstein, "Line-line intersection," disponible en MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Line-LineIntersection.html[Último acceso: 5/Febrero/2018].

[28] A. S. Glassner, Ed., *Graphics Gems 1.* San Diego: Morgan Kaufmann, 1990, ch. 5, disponible en http://proquest.safaribooksonline.com/book/illustration-and-graphics/9780080507538.

[29] J. O'Rourke, *Computational Geometry in C*, ii ed. Cambridge University Press, 1998, ch. 7. Search and intersection, disponible en http://proquest.safaribooksonline.com/book/geometry/9781107263901.

[30] "Información sobre gnu/linux en wikipedia," http://es.wikipedia.org/wiki/GNU/Linux [Último acceso 8/Febrero/2018].

[31] "Libfreenect2, página de descarga del driver (habilita la comunicación de la kinect con linux)," https://github.com/OpenKinect/libfreenect2 [Último acceso 26/Enero/2018].

[32] "Nite, alternativa de descarga (v2.2.0.5, versión utilizada en este tfg)," https://bitbucket.org/kaorun55/openni-2.2/src/2f54272802bfd24ca32f03327fbabaf85ac4a5c4/NITE%202.2%20%CE%B1/?at=master [Último acceso 26/Enero/2018].

Apéndice A

Pliego de condiciones

Para el correcto funcionamiento del sistema desarrollado en este trabajo es necesario que los equipos utilizados dispongan de unos requisitos mínimos de hardware y software.

A.1 Requisitos de Hardware

El equipo que se utilice para la ejecución del software desarrollado debe disponer de, como mínimo:

- Procesador de doble núcleo (dual core) de 3.1 GHz y 64 bit (x64).
- 4 GB de memoria.
- Puerto USB 3.0 para la conexión de la kinect.
- Cámara kinect v2.
- Al menos 1GB de memoria libre para la instalación de las librerías requeridas y los archivos requeridos para la ejecución del proyecto.

A.2 Requisitos de Software

Para el correcto funcionamiento del código se requiere:

- Sistema operativo Linux Ubuntu 16.04 LTS [30]
- Librería Libfreenect2 [31]
- Librería OpenNI 2.2.0.33 [5, 25]
- Librería NiTE 2.2.0.5 (o versión más reciente) [32]
- Librería OpenGL 3.0 [11]
- GNU GCC, NetBeans o similar.

Apéndice B

Presupuesto

B.1 Costes de equipamiento.

- Equipamiento Hardware empleado:

Concepto	**Cantidad**	**Coste unitario**	**Subtotal**
MiniPC Intel NUC 5i7RYH	1	467 €	467 €
Kinect 2	1	300 €	300 €
Coste Total			767 €

Tabla B.1: Coste del equipamiento hardware empleado.

- Recursos Software empleados:

Concepto	**Cantidad**	**Coste unitario**	**Subtotal**
Ubuntu 16.04 LTS	1	0 €	0 €
Librería Libfreenect2	1	0 €	0 €
Librería OpenNI 2	1	0 €	0 €
Librería NiTE 2 2	1	0 €	0 €
Librería OpenGL 3.0	1	0 €	0 €
NetBeans	1	0 €	0 €
Texmaker	1	0 €	0 €
Coste total			0 €

Tabla B.2: Costes de los recursos Software empleados.

B.2 Costes de mano de obra.

Concepto	Cantidad	Coste unitario	Subtotal
Desarrollo Software	320h	60€/hora	19200€
Mecanografiado del documento	80h	15€/hora	1200€
Coste total			20400€

Tabla B.3: Costes asociados a la mano de obra empleada.

B.3 Costes totales.

Concepto	Subtotal
Equipamiento Hardware	767€
Recursos Software	0€
Mano de obra	20400€
Total	21167€

Tabla B.4: Coste total del presupuesto.

El importe total del presupuesto asciende a la cantidad de VEINTIÚN MIL CIENTO SESENTA Y SIETE EUROS

En Alcalá de Henares a 6 de Abril de 2018

Alba Fortes Martínez

Graduada en Ingeniería en Electrónica y Automática Industrial

Apéndice C

Manual de instalación

Este manual indica los pasos a seguir para la correcta instalación de los componentes software necesarios para el correcto funcionamiento del código desarrollado. Así como, aquellos necesarios para su compilación y ejecución.

C.1 Instalación prerrequisitos software

Para que la aplicación desarrollada pueda ejecutarse correctamente, es necesaria la previa instalación de ciertas librerías, especificadas en el apéndice A. A continuación se describe como instalar las librerías más específicas de este proyecto, siendo estas *libfreenect2*, *OpenNI2* y *NiTE2*.

C.1.1 Libfreenect2

Libfreenect2 es una librería que actúa como driver entre la kinect y sistemas operativos distintos a Windows, como Ubuntu, permitiendo la comunicación entre ellos. A continuación se describen los pasos a seguir para su instalación por terminal.

1. Descargar la librería desde la página [31].
   git clone https://github.com/OpenKinect/libfreenect2.git
   cd libfreenect2

2. Verificar que los prerrequisitos de la librería hayan sido instalados con antelación, de no ser así instalarlos antes de proceder a la instalación de *libfreenect2*. Las librerías que se deben tener instaladas son:

   - **Build tools**
     sudo apt-get install build-essential cmake pkg-config

   - **Libusb**
     sudo apt-get install libusb-1.0.0-dev

   - **TurboJPEG**
     sudo apt-get install libturbojpeg libjpeg-turbo8-dev

- **OpenGL**
  sudo apt-get install libglfw3-dev

- **OpenNI2**
  sudo apt-get install libopenni2-dev

3. Crear un directorio donde instalar la librería, para la explicación se considerará que se ha creado un directorio *build*. Una vez instalada, cambiar el archivo `udev` por el proporcionado, y reconectar la kinect para que se actualice.
   mkdir build
   cd build
   cmake .. -DBUILD_OPENNI2_DRIVER=ON
   make
   sudo make install
   sudo ldconfig /usr/local/lib/

   sudo cp ../platform/linux/udev/90-kinect2.rules /etc/udev/rules.d

4. Verificar la correcta instalación mediante el uso de los ejemplos proporcionados con ella, como `Protonect`.

C.1.2 OpenNI2

Para la instalación de OpenNI2 se debe:

1. Descargar la librería desde [5].
2. Verificar que los prerrequisitos hayan sido previamente instalados, siendo estos:

   - GCC 4.x
     sudo apt-get install g++

   - Python 2.6+/3.x
     sudo apt-get install python

   - LibUSB 1.0.x
     sudo apt-get install libusb-1.0.0-dev

   - LibUDEV
     sudo apt-get install libudev-dev

   - Oracle Java 8
     sudo add-apt-repository ppa:webupd8team/java
     sudo apt-get update
     sudo apt-get install oracle-java8-installer

- FreeGLUT3
  sudo apt-get install freeglut3-dev

- Doxygen
  sudo apt-get install doxygen

- GraphViz
  sudo apt-get install graphviz

3. Acceder al directorio correspondiente e instalar la librería.
   chmod +x install.sh
   sudo ./install.sh

4. Copiar la librería previamente instalada en C.1.1, `libfreenect2-openni2.so`, al directorio que se acaba de instalar *../Samples/Bin/OpenNI2/Drivers*.

5. Verificar la correcta instalación mediante la ejecución de los ejemplos proporcionados, por ejemplo `NiViewer`.

C.1.3 NiTE2

Una vez instalados *libfreenect2* y *OpenNI2*, y verificado su funcionamiento, se procede a la instalación de NiTE, para lo cual se deben seguir los siguientes pasos:

1. Descargar el archivo desde [32].

2. Acceder al directorio correspondiente y proceder a la instalación de la librería.
   chmod +x install.sh
   sudo ./install.sh

3. Copiar la librería previamente instalada en C.1.1, `libfreenect2-openni2.so`, al directorio que se acaba de instalar *../Samples/Bin/OpenNI2/Drivers*.

4. Verificar la correcta instalación mediante la ejecución de los ejemplos proporcionados, por ejemplo `UserViewer`.

C.2 Compilación y ejecución del código desarrollado

La librería desarrollada se encuentra en la carpeta **libtracking**, incluida dentro de la carpeta **ispaceVirtualAlba**. Dentro de estas se encuentran los archivos necesarios para la compilación y ejecución de la aplicación, los más importantes de estos archivos son:

- `ispaceVirtual.c`: localizado en la carpeta *ispaceVirtualAlba*, se encarga de desarrollar el entorno virtual sobre el que se trabaja y con el que se interactúa. Es desde este archivo desde donde se realizan las llamadas a las funciones desarrolladas en el archivo `skeltrack.c`.

- `skeltrack.c y skeltrack.h`: localizados en la carpeta *libtracking*, contienen el algoritmo desarrollado para el cumplimiento del objetivo de este TFG, siendo este la identificación y seguimiento del esqueleto de los usuarios presentes.

En cada una de las carpetas previamente mencionadas existe un archivo `Makefile` que se encarga de compilar el programa y generar el archivo ejecutable correspondiente, en este caso solo es necesario compilar el archivo `ispaceVirtual.c` que generará el ejecutable `ispaceVirtual`. Esta compilación, así como la ejecución, se puede realizar tanto desde un entorno de desarrollo que permita la compilación por archivo Makefile, como *Netbeans*, o directamente desde terminal a través del comando *make*.

Además de estos archivos, se requiere de uno más, un archivo `*.sim`, que será el que contenga la descripción de todas las superficies del espacio representado en el entorno virtual. Este archivo estará contenido en la carpeta **IspaceRoom**, accesible desde */far-field/environments/IspaceRoom/*, siendo este directorio (*far-field*) el que contiene a su vez las librerías de soporte del GEINTRA necesarias para los archivos mencionados previamente.

Para la ejecución del algoritmo, se debe lanzar la aplicación, ejecutando el archivo `ispaceVirtual` que se ha obtenido con la compilación e indicándole a este el entorno `*.sim` al que debe llamar. Si se opta por realizar la ejecución desde terminal, el comando correspondiente debería ser, por ejemplo: *$ ./ispaceVirtual -v ../../far-field/environments/IspaceRoom/ispace.sim*. Si se opta por realizarla desde un entorno de desarrollo se debe indicar el archivo correspondiente y su ubicación como parte de las propiedades del proyecto.

Para la ejecución de este proyecto, se van a utilizar 2 archivos `.sim`, el mostrado en el ejemplo anterior (`ispace.sim`) que representa el laboratorio del grupo GEINTRA, y `idiapRoom-AV163-onlyWalls.sim`, que es un espacio reducido desarrollado para la verificación del algoritmo de colisión.

Apéndice D

Manual de usuario

Este manual presenta la interfaz del software desarrollado, proporcionando una guía de los controles que activan los distintos eventos asociados a su ejecución para facilitar su manejo, así como, una explicación sobre como modificar la posición de la cámara en el entorno virtual.

D.1 Información básica del software

Todas las funciones aquí implementadas han sido desarrolladas en el lenguaje de programación C/C++, funciones que a su vez hacen uso de las bibliotecas siguientes.

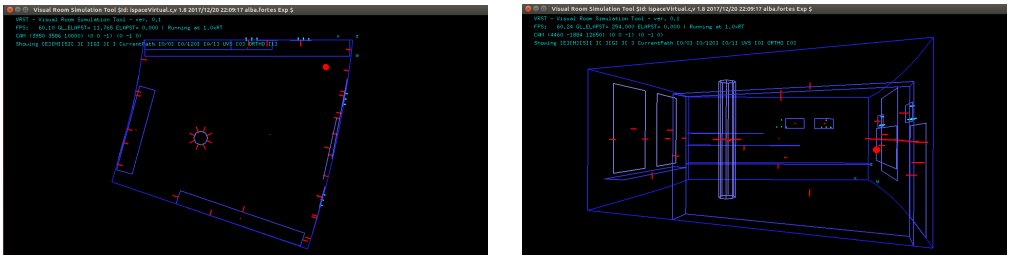
- **OpenGL**: para la generación de los gráficos 3D.
- **OpenNI**: actúa de intermediario entre la cámara Kinect y la aplicación desarrollada, permitiéndole a esta última el acceso a los datos adquiridos por la cámara.
- **NiTE**: middleware que interactúa con OpenNI y permite la interpretación de los datos de la cámara.

Además de este software, también es necesaria la instalación de la librería *libfreenect2*. Esta librería habilita la comunicación del dispositivo kinect con entornos que no sean Windows, como Ubuntu, que es sobre el que se ha trabajado en este proyecto. En específico se ha utilizado Ubuntu 16.04 LTS. Para una explicación detallada de los pasos a seguir para su instalación ver el anexo *Manual de instalación* (Apéndice C).

Para el funcionamiento del algoritmo, se debe verificar que la cámara kinect haya sido conectada al puerto USB 3.0 del ordenador.

D.2 Guía de referencia de los controles de la librería

Una vez ejecutado el archivo, aparecerá la siguiente ventana en pantalla:



(a) en 2 dimensiones. (b) en 3 dimensiones.

Figura D.1: Ventana del entorno virtual

cuando se solicite la ventana de datos de visualización, en la esquina superior derecha de la ventana que se muestra en la figura D.1, aparecerá otra donde se mostrarán los datos obtenidos por la cámara.

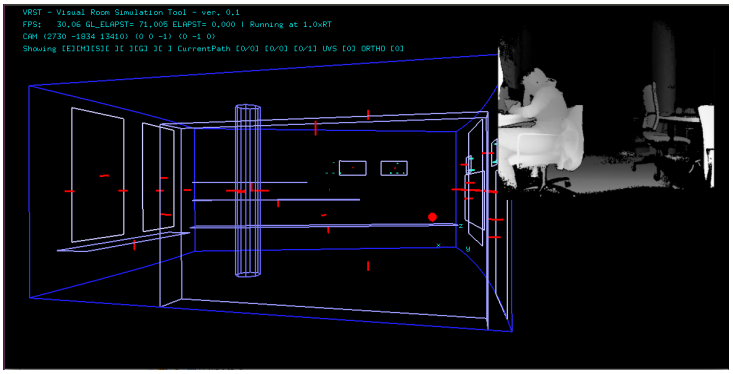


Figura D.2: Ventana del entorno virtual con ventana de visualización de datos activa.

Es sobre la figura anterior (D.2) sobre la que se definen los controles para los eventos asociados a la librería. Estos eventos se definen en función de sobre que ventana se encuentre el usuario, de esta forma se tiene:

- En la ventana principal: todos los eventos aquí mencionados están asociados al teclado.

  D Su pulsación crea o destruye la ventana de visualización de datos.

  9 Modifica el modo de calibración del esqueleto del usuario, cambia de calibración automática a requerir que el usuario se encuentre en una pose específica (PSI) para iniciarla, o viceversa. Por defecto, se activa en calibración automática.

  A Actualiza la posición de la pelota a unas coordenadas prefijadas.

- En la ventana de visualización de datos (esquina superior derecha): todos los eventos aquí mencionados están asociados al ratón del ordenador.

  – **Clic derecho** Una pulsación del botón derecho del ratón reduce el tamaño de la ventana, cuando se alcance el mínimo permitido la aplicación lo avisará por un mensaje en el terminal.

  – **Clic izquierdo** Una pulsación del botón izquierdo del ratón amplía la ventana, cuando se alcance el máximo permitido la aplicación lo avisará por un mensaje en el terminal.

Los controles asociados al entorno virtual utilizado se describen en el archivo `README` presente en la carpeta *ispaceVirtualAlba*.

D.3 Posicionamiento de la cámara en el entorno virtual

Para el posicionamiento de la kinect en el entorno virtual, tal como se ha explicado en el capítulo 3 de este documento, se ha empleado el método de las matrices de transformación homogénea. Por ello, la posición de la kinect queda determinada por los siguientes datos:

- Vector de traslación, que representa la ubicación de la kinect dentro del espacio del entorno virtual.
- Proyección de los ejes del sistema de referencia de la kinect sobre el sistema de referencia del entorno virtual, que representan la matriz de rotación.

Ambos datos se especifican en la cabecera del archivo `skeltrack.c`, mediante el uso de las variables $kinectPos$, x_kinect, y_kinect y z_kinect, siendo estas tres últimas las que definen los ejes de la kinect. A continuación se proporciona un ejemplo de la modificación de estas variables.

En las figuras D.3 y D.4, se representan 2 posiciones distintas de la cámara respecto al entorno virtual, donde los ejes verdes representan el sistema de referencia del entorno y los azules el de la kinect. En ellas se han fijado unas traslaciones (a, b, c) y (a', b', c') respecto al sistema de referencia del entorno. En el código quedaría:

- **Posición 1:** la kinect se sitúa con su eje X en sentido opuesto al eje X del entorno, su eje Z paralelo al eje Y del entorno, y su eje Y paralelo al eje Z del entorno.

```
// Traslación
TPoint kinectPos={a, b, c};

// Rotación
TPoint x_kinect={-1, 0, 0};
TPoint y_kinect={0, 0, 1};
TPoint z_kinect={0, 1, 0};
```

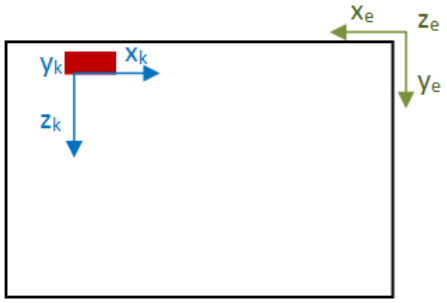


Figura D.3: Cámara kinect en posición 1.

- **Posición 2:** la kinect se sitúa con su eje X formando 240º en sentido antihorario con el eje X del entorno, su eje Z formando 60º en sentido antihorario con el eje Y del entorno, y su eje Y paralelo al eje Z del entorno.

```
// Traslación
TPoint kinectPos={a', b', c'};

// Rotación
TPoint x_kinect={-0.5, -0.866, 0};
TPoint y_kinect={0, 0, 1};
TPoint z_kinect={-0.866, 0.5, 0};
```

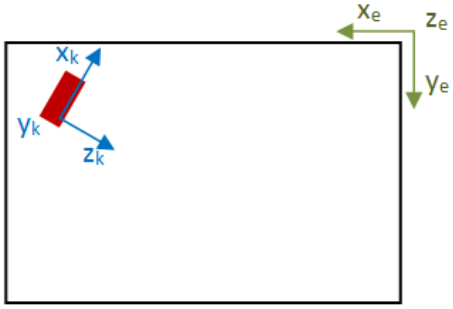


Figura D.4: Cámara kinect en posición 2.

En estos ejemplos, las proyecciones de los ejes de la kinect se definen como unitarios para el caso de la rotación, pero no sería necesario ya que el código desarrollado los vuelve unitarios antes de operar con ellos.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá